Data-Oriented Design

Richard Fabian

September 26, 2013

Contents

Introduction

It took a lot longer than expected to be able to write this book, but it's been all the better for how long the ideas have been fermenting in my head and how many of the concepts have been tested out in practice. In many cases, the following chapters are ideas that started out as simple observations and slowly evolved into solid frameworks for building software in a data-oriented manner. Some sections have remained largely unchanged from their first drafts, such as the sections on existence based processing or the arguments against the object-oriented approach, but others have been re-written a number of times, trying to distill just the right kind of information at the right pace.

Most people come at data-oriented design from object-oriented design, and have heard of it only because of people claiming that object-oriented design is bad, or wrong, or simply not the only way of doing things. This may be the case for large scale software, and though object-oriented code does have its place (as we shall discuss in chapter 15), it has been the cause[1] of much wasted time and effort during its relatively short life in our passionate industry of software development and games development in particular.

My own journey through imperative procedural programming, then Object-oriented programming then finally finding, embracing,

[1] Large Scale C++ is a book almost entirely dedicated to showing how object oriented development needn't destroy the productivity of large scale software projects

and now spreading the word of Data-Oriented design, all started with C++. I like to think of C++ as a go to language for the best of both worlds when you require one of the worlds to be assembly level quality of a control over your instructions. The other world is the world of abstractions, the ability to create more code that does more in less time. Over the years, I have learned a great deal about how C++ helps add layer upon layer of abstraction to help make less code do more, but I've also seen how the layers of various quality code can cause a cascade of errors and unmaintainable spaghetti that puts off even the most crunch hardened programmers.

I'd like to thank those who have helped in the making of this book, whether by reading early versions of this text and criticising the content of structure, or by being an inspiration or guiding light on what would truly best represent this new paradigm. You can thank the critics for the layout of the chapters and the removal of so much of the negativity that originally sat on the pages in the critique of design patterns. Design patterns are very nearly the opposite of data-oriented design; it was natural that they took a beating, but the whole chapter felt like a flame war so was removed.

This book is a practical guide for serious game developers. It is for game developers working to create triple A titles across multiple platforms, for independent developers trying to get the most out of their chosen target hardware, in fact for anyone who develops cutting edge software in restrictive hardware. It is a book about how to write code. It is a book written to educate games developers in a coding paradigm that is future proof, unlike the style of coding we've become so accustomed to. It is a book rooted in C++, the language of choice by games developers of the last ten years, and provides practical advice on how to migrate without throwing away years of accumulated code and experience. This book is about how you can transform your development.

If you're a diehard object-oriented programmer who doesn't believe all this hype around this new pseudo-paradigm, then you've probably not even picked up this book, but for those that do, I hope that this book will give you the chance to change your mind. So

here's to opening minds, but not so much that your brains fall out.

Chapter 1

Data-Oriented Design

Data-oriented design has been around for decades in one form or another, but was only officially given a name by Noel Llopis in his September 2009 article of the same name. The idea that it is a programming paradigm is seen as contentious as many believe that it can be used side by side with another paradigm such as object-oriented programming, procedural programming, or functional programming. In one respect they are right, data-oriented design can function alongside the other paradigms, but so can they. A Lisp programmer knows that functional programming can coexist with object-oriented programming and a C programmer is well aware that object-oriented programming can coexist with procedural programming. We shall ignore these comments and claim that data-oriented design is another important tool; a tool just as capable of coexistence as the rest.

The time was right in 2009. The hardware was ripe for a change in how to develop. Badly programmed potentially very fast computers, hindered by a hardware ignorant programming paradigm made many engine programmers weep. The times have changed, and many mobile and desktop solutions now need the data-oriented design approach less, not because the machines are better at mitigat-

ing ineffective programming, but the games being designed are less demanding. As we move towards ubiquity for multi-core machines, not just the desktops, but our phones too, and towards a world of programming for massively parallel processing design as in the compute cores on our graphics cards and in our consoles, the need for data-oriented design will only grow. It will grow because abstractions and serial thinking will be the bottleneck of your competitors, and those that embrace the data-oriented approach will thrive.

1.1 It's all about the data

Data is all we have. Data is what we need to transform in order to create a user experience. Data is what we load when we open a document. Data is the graphics on the screen and the pulses from the buttons on your game pad and the cause of your speakers and headphones producing waves in the air and the method by which you level up and how the bad guy knew where you were to shoot at you and how long the dynamite took to explode and how many rings you dropped when you fell on the spikes and the current velocity of every particle in the beautiful scene that ended the game, that was loaded off the disc and into your life. Any application is nothing without its data. Photoshop without the images is nothing. Word is nothing without the characters. Cubase is worthless without the events. All the applications that have ever been written have been written to output data based on some input data. The form of that data can be extremely complex, or so simple it requires no documentation at all, but all applications produce and need data.

Instructions are data too. Instructions take up memory, use up bandwidth, and can be transformed, loaded, saved and constructed. It's natural for a developer to not think of instructions as being data[1], but there is very little differentiating them on older, less protective hardware. Even though memory set aside for executables is

[1] unless they are a Lisp programmer

protected from harm and modification on most contemporary hardware, this relatively new invention is still merely an invention. Instructions are still data, and once more, they are what we transform too. We take instructions and turn them into actions. The number, size, and frequency of them is something that matters and that we have control over.

This forms the basis of the argument for a data-oriented approach to development, but leaves out one major element. All this data and the transforming of data, from strings, to images, to instructions, they all have to run on something. Sometimes that thing is quite abstract, such as a virtual machine running on unknown hardware. Sometimes that thing is concrete, such as knowing which specific CPU and what speed and memory capacity and bandwidth you have available. But in all cases, the data is not just data, but data that exists on some hardware somewhere, and it has to be transformed by some hardware. In essence, data-oriented design is the practice of designing software by developing transforms for well formed data where well formed is guided by the target hardware and the transforms that will operate on it. Sometimes the data isn't well defined, and sometimes the hardware is equally evasive, but in most cases a good background of hardware appreciation can help out almost every software project.

If the ultimate result of an application is data, and all input can be represented by data, and it is recognised that all data transforms are not performed in a vacuum, then a software development methodology can be founded on these principles, the principles of understanding the data, and how to transform it given some knowledge of how a machine will do what it needs to do with data of this quantity, frequency, and it's statistical qualities. Given this basis, we can build up a set of founding statements about what makes a methodology data-oriented.

1.2 Data is not the problem domain

The first principle: Data is not the problem domain.

For some, it would seem that data-oriented design is the antithesis of most other programming paradigms because data-oriented design is a technique that does not readily allow the problem domain to enter into the software so readily. It does not recognise the concept of an object in any way, as data is consistently without meaning, whereas the abstraction heavy paradigms try to pretend the computer and its data do not exist at every turn, abstracting away the idea that there are bytes, or CPU pipelines, or other hardware features.

The data-oriented design approach doesn't build the real world problem into the code. This could be seen as a failing of the data-oriented approach by veteran object-oriented developers, as many examples of the success of object-oriented design come from being able to bring the human concepts to the machine, then in this middle ground, a solution can be written in this language that is understandable by both human and computer. The data-oriented approach gives up some of the human readability by leaving the problem domain in the design document, but stops the machine from having to handle human concepts at any level by just that same action.

Let us consider first how the problem domain becomes part of the software in programming paradigms that promote abstraction. In the case of objects, we tie meanings to data by associating them with their containing classes and their associated functions. In high level abstraction, we separate actions and data by high level concepts, which might not apply at the low level, thus reducing the likelihood that the functions will be efficiently implemented.

When a class owns some data, it gives that data a context, and that context can sometimes limit the ability to reuse the data. Adding functions to a context can bring in further data, which quickly leads to classes that contain many different pieces of data that are unrelated in themselves, but need to be in the same class because an operation required a context, but the operation also re-

quired additional data. Normally this becomes hard to untangle as functions that operate over the whole context drag in random pieces of data from all over the class meaning that many data items cannot be removed as they would then be inaccessible.

When we consider the data from the data-oriented design point of view, data is mere facts that can be interpreted in whatever way necessary to get the output data in the format it needs to be. We only care about what transforms we do, and where the data ends up. In practice, when you discard meanings from data, you also reduce the chance of tangling the facts with their contexts, and thus you also reduce the likelihood of mixing unrelated data just for the sake of an operation or two.

1.3 Data and Statistics

The second principle: Data is type, frequency, quantity, shape and probability.

The second statement is that data is not just the structure. A common misconception about data-oriented design is that it's all about cache misses. Even if it was all about making sure you never missed the cache, and it was all about structuring your classes so the hot and cold data was split apart, it would be a generally useful addition to your toolkit of thought, but data-oriented design is about all aspects of the data. To write a book on how to avoid cache misses, you need more than just some tips on how to organise your structures, you need a grounding in what is really happening inside your computer when it is running your program. Teaching that in a book is also impossible as it would only apply to one generation of hardware, and one generation of programming languages, however data oriented design is not rooted in one language and one set of hardware. The schema of the data is still important, but the actual values are as important, if not more so. It is not enough to have some photographs of a cheetah to determine how fast it can run. You need to see it in the wild.

The data-oriented design model is centred around data, live data, real data, information data. Object-oriented design is centred around the problem and its solution. Objects, not real things, but abstract representations of things that make up the design of the solution to the problem presented in the application design document. The objects only manipulate the data needed to represent them without any consideration for the hardware or the real world data patterns or quantities. This is why object-oriented design allows you to quickly build up first versions of applications, allowing you to put the first version of the design document or problem definition directly into the code.

Data-oriented design takes a different approach to the problem, instead of assuming that we know nothing about the hardware, it assumes we know nothing about the problem. Anyone who has written a sizable piece of software should recognise that the technical design for any project can change so much that there is hardly anything recognisable from the first draft in the final implementation. Data-oriented design avoids this waste of resources by never assuming that the design needs to exist anywhere other than in a document while it proceeds to provide a solution to the current problem.

Data-Oriented Design takes it's cues from the data that is seen or expected. Instead of planning for all eventualities, or planning to make things adaptable, it uses the most probable input to direct the choice of algorithm. Instead of planning to be extendible, it plans to be simple, and get the job done. Extendible can be added later, with the safety net of unit tests to ensure that it remains working as it did while it was simple.

1.4 Data changes

Data-oriented design is current. Object-oriented design starts to show its weaknesses when designs change. Object-oriented design suffers from an inertia inherent in keeping the problem domain coupled with the implementation. A data-oriented approach to design

takes note of the change in design by understanding the change in the data. The data-oriented approach to design also allows for change to the code when the source of data changes, unlike the encapsulated internal state manipulations of the object-oriented approach. In general, data-oriented design handles change better as pieces of data and transforms can be more simply coupled and decoupled than objects can be mutated and reused.

The reason this is so comes from the linking of intention and aspect that comes with lumping data and functions in with concepts of objects where the objects are the schema, the aspect, the use case, and the real world or design counterpart to the code. If you link your data manipulations to your data then you make it hard to unlink data related by operation. If you link your operations by related data, then you make it hard to unlink your operations when the data changes or splits. If you keep your data in one place, write operations in another place, and keep the aspects and roles of data intrinsic from how the operations and transforms are applied to the data, then you will find that many refactorings that would have been large and difficult in object oriented code, become trivial. With this benefit comes a cost of keeping tabs on what data is required for each operation, and the potential danger of desynchronisation. This consideration can lead to keeping some cold code in an object oriented style where objects are responsible for maintaining internal consistency over efficiency and mutability.

A big misunderstanding for many new to the data-oriented design paradigm, a concept brought over from abstraction based development, is that we can design a static library or set of templates to provide generic solutions to everything presented in this book as a data-oriented solution. The awful truth is that data, though it can be generic by type, is not generic in any real world sense. The values are different, and often contain patterns that we can turn to our advantage. How would it be possible to do compression if it were not for patterns? Our runtime code can also benefit from this but it is overlooked as being not object-oriented, or being too hard coded. It can be better to hard code a transform than pretend it's not hard

coded by wrapping it in a generic container and using the wrong algorithms on it. Using existing templates like this provide a known benefit of a minor increase in readability to those who already know of the library, and potentially less bugs if the functionality was in some way generic. But, if the functionality was not very well mapped to the existing generic solution, writing it with a templated function and then extending would possibly make the code harder to read by hiding the fact that the technique had been changed subtly. Hard coding a new algorithm is frequently a better choice as long as it has sufficient tests, and tests are easier to write if you constrain yourself to the facts about the data, and only work with simple data and not stateful objects.

1.5 How is data formed?

The games we write have a lot of data, in a lot of different formats. We have textures in multiple formats for multiple platforms. There are animations, usually optimised for different skeletons or types of playback. There are sounds, lights, and scripts. Don't forget meshes, they now consist of multiple buffers of attributes. Only a very small proportion of meshes are old fixed function type with vertices containing positions, UVs, and normals. The data in games development is hard to box, and getting harder to pin down as more ideas that were previously considered impossible have now become commonplace. This is why we spend a lot of time working on editors and tool chains, so we can take the free form output from designers and artists and find a way to put it into our engines. Without our tool-chains, editors, viewers, and tweaking tools, there would be no way we could produce a game with the time we have. The Object-oriented approach provides a good way to wrap our heads around all these different formats of data. It gives a centralised view of where each type of data belongs, and classifies it by what can be done to it. This makes it very easy to add and use data quickly, but implementing all these different wrapper objects takes

time. Adding new functionality to these objects can sometimes require large refactorings as occasionally objects are classified in such a way that they don't allow for new features to exist. For example, in many old engines, there was no way to upload a texture that didn't use a 32 bit or less pixel stride. With the advent of floating point textures all that code required a minor refactoring. In the past it was not possible to read a texture from the vertex shader, so when texture based skinning came along, many engine programmers had to refactor their render update. They had to allow for a vertex shader texture upload because it might be necessary when uploading transforms for rendering a skinned mesh. In many engines, mesh data is optimised for rendering, but when you have to do mesh ray casting to see where bullets have hit, or for doing IK, or physics, then you need multiple representations of an entity, at which point the object oriented approach starts to look cobbled together as there are less objects that represent real things, and more objects used as containers so programmers can think in larger building blocks. These blocks hinder though, as they become the only blocks used in thought, and stop potential mental connections from happening. We went from 2D sprites, to 3D meshes following the format of the hardware provider, to custom data streams and compute units turning the streams into rendered triangles. Wave data, to banks, to envelope controlled grain tables and slews of layered sounds. Tile maps, to portals and rooms, to streamed, multiple level of detail chunks of world, to hybrid mesh palette, props, and unique stitching assets. From flip-book, to euler angle sequences, to quaternions and slerped anims, to animation trees and behaviour mapping.

All these types of data are pretty common if you've worked in games at all, and many engines do provide a abstraction to these more fundamental types. When a new type of data becomes heavily used, it is promoted into the engine as a core type. We normally consider the trade off of new types being handled as special cases until they become ubiquitous, to be one of usability vs performance. We don't want to provide free access to the lesser understood elements of game development. People who are not, or can not, invest time

in finding out how best to use new features, should be discouraged from using them. The Object-Oriented games development way to do that is to not provide objects that represent them, and instead only offer the features to people who know how to utilise the more advanced tools.

Apart from the objects representing digital assets, there are also objects for internal game logic. For every game there are objects that only exist to further the gameplay. Collectable card games have a lot of textures, but they also have a lot objects for rules, card stats, player decks, match records, even objects to represent the current state of play. All of these objects are completely custom designed for one game. There may be sequels, but even they could well use quite different game logic, and therefore require different data, which would imply different methods on the now guaranteed different objects.

Game data is complex. Any first layout of the data is inspired by the game's design. However, once the development is underway, it needs to keep up with however the game evolves. Object-oriented techniques offer a quick way to implement a given design. Object-oriented development is quick at implementing each design in turn, but it doesn't offer a way to migrate from one data schema to the next. There are hacks, such as those used in version based asset handlers, but normally, games developers change the tool-chain and the engine at the same time, do a full re-export of all the assets, and commit to the next version all in one go. This can be quite a painful experience if it has to happen over multiple sites at the same time, or if you have a lot of assets, or if you are trying to provide engine support for more than one title, and only one wants to change to the new revision.

There has not yet been a successful effort to build a generic game asset solution. This is because all games differ in so many subtle ways that if you did provide a generic solution, it wouldn't be a game solution, just a new language. There is no solution to be found in trying to provide all the possible types of object that a game can use. But, there is a solution if we go back to thinking

about a game as merely running a set of computations on some data.

1.6 What can provide a computational framework for such complex data?

Games developers are notorious for thinking about games development from either a low level all out performance perspective, or from a very high level gameplay and interaction perspective. This may have come about because of the widening gap between the amount of code that has to be performant, and the amount of code to make the game complete. Object-oriented techniques provide good coverage of the high level aspect, and the assembler gurus have been hacking away at performance for so long even their beards have beards. There has never been much of a middle ground in games development, which is probably why the structure and performance techniques employed by big-iron companies never seemed useful. When games development was first flourishing in the late 1990's, academic or corporate software engineering practices were seen as suspicious because wherever they were employed, there was a dramatic drop in game performance, and whenever any prospective employees came from those industries, they never managed to impress. As games machines became more like standard micro-computers, and standard micro-computers became more similar to the mainframes of old, the more obvious it became that standard professional software engineering practices could be useful. Now the scale of games has grown to match the hardware, and with each successive generation, the number of man hours to develop a game has grown exponentially, which is why project management and software engineering practices have become de-facto at the larger games companies. There was a time when games developers were seen as cutting edge programmers, inventing new technology as the need arises, but with the advent of less adventurous hardware (most notably in the x86 based XBox, and it's only mildly interesting successor), there has been less call for ingenious coding practices, and more call for a standardised pro-

cess. This means that game development can be tuned to ensure the release date will coincide with marketing dates. There will always be an element of randomness in high profile games development because there will always be an element of innovation that virtually guarantees that you will not be able to predict how long the project, or at least one part of the project, will take.

Part of the difficulty in adding new and innovative features to a game is the data layout. If you need to change the data layout for a game, it will need objects to be redesigned or extended in order to work within the existing framework. If there is no new data, then a feature might require that previously separate systems suddenly be able to talk to each other quite intimately. This coupling can often cause system wide confusion with additional temporal coupling and corner cases so obscure they can only be found one time in a million. This sounds fine to some developers, but if you're expecting to sell five to fifty million copies of your game, that's five to fifty people who can take a video of your game, post it on the internet, and call your company rubbish because they hadn't fixed an obvious bug.

Big iron developers had these same concerns back in the 1970's. Their software had to be built to high standards because their programs would frequently be working on data that was concerned with real money transactions. They needed to write business logic that operated on the data, but most important of all, they had to make sure the data was updated through a provably careful set of operations in order to maintain its integrity. Database technology grew from the need to process stored data, to do complex analysis on it, to store and update it, and be able to guarantee that it was valid at all times. To do this, the ACID test was used to ensured atomicity, consistency, isolation, and durability. Atomicity was the test to ensure that all transactions would either complete, or do nothing. It could be very bad for a database to update only one account in a financial transaction. There could be money lost, or created if a transaction was not atomic. Consistency was added to ensure that all the resultant state changes that should happen during a transaction do happen, that is, all triggers that should fire, do, even if the

triggers cause triggers, with no limit. This would be highly important if an account should be blocked after it has triggered a form of fraud detection. If a trigger has not fired, then the company using the database could risk being liable for even more than if they had stopped the account when they first detected fraud. Isolation is concerned with ensuring that all transactions that occur cannot cause any other transactions to differ in behaviour. Normally this means that if two transactions appear to work on the same data, they have to queue up and not try to operate at the same time. Although this is generally good, it does cause concurrency problems. Finally, durability. This was the second most important element of the four, as it has always been important to ensure that once a transaction has completed, it remains so. In database terminology, durability meant that the transaction would be guaranteed to have been stored in such a way that it would survive server crashes or power outages. This was important for networked computers where it would be important to know what transactions had definitely happened when a server crashed or a connection dropped.

Modern network games also have to worry about highly important data like this, especially with non-free downloadable content, and even more so with consumable downloadable content. To provide much of the functionality required of the database ACID test, games developers have gone back to looking at how databases were designed to cope with these strict requirements and found reference to staged commits, idempotent functions, techniques for concurrent development and a vast literature base on how to design tables for a database.

Databases provide a way to store highly complex data in a structured way while providing a simple to learn language for transforming and generating reports based on that data. The language, SQL, invented in the 1970's by Donald D. Chamberlin and Raymond F. Boyce at IBM, provides a method by which it is possible to store computable data while also maintaining data relationships. But, games don't have simple computable data, they have classes and objects. Database technology doesn't work for the Object-oriented

approach that games developers use.

The data in games can be highly complex, it doesn't neatly fit into database rows. Not all objects can fit into rows of columns. Try to find the right table columns to describe a level file. Try to find the right columns to describe a car in a racing game, do you have a column for each wheel? Do you have a column for each collision primitive, or just a column for the collision mesh?

The obvious answer is that game data just doesn't fit into the database way of thinking. However, that's only because we've not normalised the data. We'll stick with a level file for a while as we find out how years old techniques can provide a very useful insight into what game data is really doing. There are some types of data where it doesn't make sense to move into databases, such as the raw assets (sounds, textures, vertex buffers etc.) but these can be seen as primitives, much like the integers, floating point numbers, strings and boolean values we shall be working with.

What we shall discover is that everything we did was already in a database, but it wasn't obvious to us because of how we store our data. The structure of our data is a trade-off against performance, readability, maintenance, future proofing, extendibility and reuse. The most flexible database in common use is the file-system. It has one table with two columns. A primary key of the file path, and a string for the data. This simplest database system is the perfect fit for a completely future proof system. The more complex the tables get, the less future proof, and the less maintainable, but the higher the performance and readability. For example, a file has no documentation of its own, but the schema of a database could be all that is required to understand a sufficiently well designed database. That's how games don't even appear to have databases. They are so complex for the sake of performance that they have forgotten that they are merely a data transform.

Chapter 2

Existence Based Processing

If you saw that there weren't any apples in stock, would you still haggle over their price?

2.1 Why use an if

When studying software engineering you may find reference to cyclomatic complexity or conditional complexity. This is a complexity metric providing a numeric representation of the complexity of code, and is used in analysing large scale software projects. Cyclomatic complexity concerns itself only with flow control. The formula, summarised for our purposes, is one (1) plus the number of conditionals present in the system being analysed. That means for any system it starts at one, and for each if, while, for, and do-while, we add one. We also add one per path in a switch statement excluding the default case if present (this is because whether or not the default case is included, it is part of the possible routes through the code, so counts like the else case of an if, as in, it isn't counted, only the

met conditions are counted.) under the hood, if we consider how a virtual call works, that is, a lookup in a function pointer table followed by a branch into the class member function, we can see that a virtual call is effectively just as complex as a switch statement. Counting the flow control statements is more difficult in a virtual call because to know the complexity value, you have to know the number of possible classes member functions that can fulfil that request. In the case of a virtual call, you have to count the number of overrides to a base virtual call. If the base is pure-virtual, then you may subtract one from the complexity. However, if you don't have access to all the code that is running, which can be possible in the case of dynamically loaded libraries, then the number of different code paths that the process can take increases by an unknown amount. This hidden or obscured complexity is necessary to allow third party libraries to interface with the core process, but requires a level of trust that implies that no single part of the process is ever going to be thoroughly tested.

Analysing the complexity of a system helps us understand how difficult it is to test, and in turn, how hard it is to debug. Sometimes, the difficulty we have in debugging comes from not fully observing all the flow control points, at which point the program will have entered into a state we had not expected or prepared for. With virtual calls, the likelihood of that happening can dramatically increase as we can not be sure that we know all the different ways the code can branch until we either litter the code with logging, or step through in a debugger to see where it goes at run-time.

Returning to complexity in general, we must submit that we use flow control to change what is executed in our programs. In most cases these flow controls are put in for one of two reasons: design, and implementation necessity. The first is when we need to implement the design, a gameplay feature that has to only happen when some conditions are met, such as jumping when the jump button is pressed, or autosaving at a save checkpoint when the savedata is dirty. The second form of flow control is structural, or defensive programming techniques where the function operating on the data isn't

sure that the data exists, or is making sure bounds are observed.

In real world cases, the most common use of an explicit flow control statement is in defensive programming. Most of our flow control statements are just to stop crashes, to check bounds, pointers being null, or any other exceptional cases that would bring the program to a halt. The second most common is loop control, though these are numerous, most CPUs have hardware optimisations for these, and most compilers do a very good job of removing condition checks that aren't necessary. The third most common flow control comes from polymorphic calls, which can be helpful in implementing some of the gameplay logic, but mostly are there to entertain the do-more-with-less-code development model partially enforced in the object-oriented approach to writing games. Actual visible game design originating flow control comes a distant fourth place in these causes of branching, leading to an under-appreciation of the effect each conditional has on the performance of the software. That is, when the rest of your code-base is slow, it's hard to validate writing fast code for any one task.

If we try to keep our working set of data as a collections of arrays, we can guarantee that all our data is not null. That one step alone will eliminate most of our flow control statements. The third most common set of flow control statements were the inherent flow control in a virtual call, they are covered later in section 2.5, but simply put, if you don't have an explicit type, you don't need to switch on it, so those flow control statements go away as well. Finally, we get to gameplay logic, which there is no simple way to eradicate. We can get most of the way there with condition tables which will be covered in chapter 5, but for now we will assume they are allowed to stay.

Reducing the amount of conditions, and thus reducing the cyclomatic complexity on such a scale is an amazing benefit that cannot be overlooked, but it is one that comes with a cost. The reason we are able to get rid of the check for null is that we now have our data in a format that doesn't allow for null. This inflexibility will prove to be a benefit, but it requires a new way of processing our entities.

Where we once had rooms, and we looked in the rooms to find out if there were any doors on the walls we bumped into (in order to either pass through, or instead do a collision response,) we now look in the table of doors to see if there are any that match our roomid. This reversal of ownership can be a massive benefit in debugging, but sometimes can appear backwards when all you want to do is find out what doors you can use to get out of a room.

If you've ever worked with shopping lists, or todo lists, you'll know how much more efficient you are when you have a definite list of things to get or get done. It's very easy to make a list, and easy to add to it too. If you're going shopping, it's very hard to think what might be missing from your house in order to get what you need. If you're the type that tries to plan meals, then a list is nigh on essential as you figure out ingredients and then tally up the number of tins of tomatoes, or how many different meats or vegetables you need to last through all the meals you have planned. If you have a todo list and a calendar, you know who is coming and what needs to be done to prepare for them, how many extra mouths need feeding, how many extra bottles of beer to buy, or how much washing you need to do to make enough beds for the visitors. Todo lists are great because you can set an end goal and then add in sub tasks that make a large and long distant goal seem more doable, and also provide a little urgency that is usually missing when the deadline is so far away.

When your program is running, if you don't give it lists to work with, and instead let it do whatever comes up next, it will be inefficient, slow, and probably have irregular frame timings. Inefficiency comes from not knowing what kind of processing is coming up next. In the case of large arrays of pointers to heterogeneous classes all being called with an `update()` function, you can get very high counts of cache misses both in data and instruction cache. If the code tries to do a lot with this pointer, which it usually does because one of the major beliefs of object-oriented programmers is that virtual calls are only for higher level operations, not small, low level ones, then you can virtually guarantee that not only will the instruction and

data cache be thrashed by each call, but most branch predictor slots could be too dirty to offer any benefit when the next `update()` runs. Assuming that virtual calls don't add up because they are called on high level code is fine until they become the go to programming style and you stop thinking about how they affect your application when there are millions of virtual calls per second. All those inefficient calls are going to add up somewhere, but luckily for the object oriented zealot, they never appear on any profiles. They always appear somewhere in the code that is being called. Slowness also comes from not being able to see how much work needs to be done, and therefore not being able to scale the work to fit what is possible in the time-frame given. Without a todo list, and an ability to estimate the amount of time that each task will take, it is impossible to decide the best course of action to take in order to reduce overhead while maintaining feedback to the user. Irregular frame timings also can be blamed on not being able to act on distant goals ahead of time. If you know you have to load an area because a player has ventured into a position where it is possible they will soon be entering an unloaded area, the streaming system can be told to drag in any data necessary. In most games this happens with explicit triggers, but there is no such system for many other game elements. It's unheard of for an AI to pathfind to some goal because there might soon be a need to head that way. It's not commonplace to find a physics system doing look ahead to see if a collision has happened in the future in order to start doing a more complex breakup simulation. But, if you let your game generate todo lists, shopping lists, distant goals, and allow for preventative measures by forward thinking, then you can simplify your task as a coder into prioritising goals and effects, or writing code that generates priorities at runtime.

In addition to the obvious lists, metrics on your game are highly important. If you find that you've been optimising your game for a silky smooth frame rate and you think you have a really steady 30fps or 60fps, and yet your customers and testers keep coming back with comments about nasty frame spikes and dropout, then you're not profiling the right thing. Sometimes you have to profile a game

while it is being played. Get yourself a profiler that runs all the time, and can report the state of the game when the frame time goes over budget. Sometimes you need the data from a number of frames around when it happened to really figure out what is going on, but in all cases, unless you're letting real testers run your profiler, you're never going to get real world profiling data.

Existence-based-processing is when you process every element in a homogeneous set of data. You run the same instructions for every element in that set. There is no definite requirement for the output in this specification, however, usually it is one of three types of operation: filter, mutation, or emission. A mutation is a one to one manipulation of the data, it takes incoming data and some constants that are setup before the transform, and produces one element for each input element. A filter takes incoming data, again with some constants set up before the transform, and produces one element or zero elements for each input element. An emission is a manipulation on the incoming data that can produce multiple output elements. Just like the other two transforms, an emission can use constants, but there is no guaranteed size of the output table; it can produce anywhere between zero and infinity elements.

Every CPU can efficiently handle running processing kernels over homogeneous sets of data, that is, doing the same operation over and over again over contiguous data. When there is no global state, no accumulator, it is proven to be parallelisable. Examples can be given from existing technologies such as mapreduce and opencl as to how to go about building real work applications within these restrictions. Stateless transforms also commit no crimes that prevent them from being used within distributed processing technologies. Erlang relies on these as language features to enable not just thread safe processing, not just inter process safe processing, but distributed computing safe processing. Stateless transforms of stateful data is highly robust, and deeply parallelisable.

Within the processing of each element, that is for each datum operated on by the transform kernel, it is fair to use control flow. Almost all compilers should be able to reduce simple local value

branch instructions into a platform's preferred branchless representation. When considering branches inside transforms, it's best to compare to existing implementations of stream processing such as graphics card shaders or OpenCL kernels.

In predication, flow control statements are not nearly ignored, they are used instead as an indicator of how to merge two results. When the flow control is not based on a constant, a predicated if will generate code that will run both sides of the branch at the same time and discard one result based on the value of the condition. It manages this by selecting one result based on the condition, in some CPUs there is an fsel intrinsic, other CPUs may have a cmov instruction, but all CPUs can use masking to effect this trick.

There are other solutions in the form of SIMD and MIMD. SIMD or single-instruction-multiple-data allows the parallel processing of data when the instructions are the same. If there are any conditionals, then as long as the result of the condition is the same across the data, the function returns quickly. If the condition result is different across the different data, the function stalls for one branch remembering which of the data had chosen which path, completes for one side, then goes back and continues for the other side. This costs time, but is not as expensive as predication as it does not always run both branches. In other systems, the condition can lead to new micro jobs handed off to different cores, converging back to the main thread only once all branches have completed.

In MIMD, that is multiple instruction, multiple data, every piece of data can be operated on by a different set of instructions. Each piece of data can take a different path. This is the simplest to code for because it's how most parallel programming is currently done. We add a thread and process some more data with a separate thread of execution. MIMD includes multi-core general purpose CPUs. MIMD often allows shared memory access and all the synchronisation issues that come with it. MIMD is by far the easiest to get up and running, but it is also the most prone to the kind of rare fatal error that costs a lot of time to debug.

2.2 Don't use booleans

When you study compression technology, one of the most important aspects you have to understand is the different between data and information. There are many ways to store information in systems, from literal strings that can be parsed to declare that something exists, right down to something simple like a single bit flag to show that a thing might have an attributes. Examples include the text that declares the existence of a local variable in a scripting language, or the bit field that contains all the different collision types a physics mesh will respond to. Sometimes we can store even less information than a bit by using advanced algorithms such as arithmetic encoding, or by utilising domain knowledge. Domain knowledge normalisation applies in most games development, but it is very infrequently applied. As information is encoded in data, and the amount of information encoded can be amplified by domain knowledge, it's important that we begin to see the advice offered by compression techniques is that: all we are really encoding is probabilities.

If we take an example, a game where the entities have health, regenerate after a while of not taking damage, can die, can shoot each other, then lets see what domain knowledge can do to reduce processing.

We assume this domain knowledge: if you have full health, then you don't need to regenerate. Once you have been shot, it takes some time until you begin regenerating. Once you are dead, you cannot regenerate. Once you are dead you have zero health.

If we have a table for the entity thus:

```
1  struct entity {
2    // information about the entity position
3    // ...
4    // now health data in the middle of the entity
5    float timeoflastdamage;
6    float health;
7    // ...
8    // other entity information
9  };
10 list<entity> entities;
```

Listing 2.1: naive entity table

Then we can run an update function over the table that might look like this:

```
void updatehealth( entity *e ) {
  timetype timesincelastshot = e->timeoflastdamage - currenttime;
  bool ishurt = e->health < max_health && e->health > 0;
  bool regencanstart = timesincelastshot >
       time_before_regenerating;
  if( ishurt && regencanstart ) {
    e->health = min(max_health, e->health + ticktime * regenrate);
  }
}
```

Listing 2.2: every entity health regen

Which will run for every entity in the game, every update.

We can make this better by looking at the flow control statement. The function only needs to run if the health is less than full health, and more than zero. The regenerate function only needs to run if it has been long enough since the last damage dealt.

Let's change the structures:

```
struct entity {
  // information about the entity position
  // ...
  // other entity information
};
struct entitydamage {
  float timeoflastdamage;
  float health;
}
list<entity> entities;
map<entityref,entitydamage> entitydamages;
```

Listing 2.3: existence based processing style health

We can now run the update function over the health table rather than the entities.

```
void updatehealth() {
  foreach( eh in entitydamages ) {
    entityref entity = eh->first;
```

```
4       entitydamage &ed = eh->second;
5       if( ed.health < 0 ) {
6         deadentities.insert( entity );
7         discard(eh);
8       } else {
9         timetype timesincelastshot = eh->timeoflastshot -
              currenttime;
10        bool regencanstart = timesincelastshot >
              time_before_regenerating;
11        if( regencanstart )
12          eh->health =eh->health + ticktime * regenrate;
13        if( eh->health > max_health )
14          discard(eh);
15      }
16    }
17  }
```

Listing 2.4: every entity health regen

We only add a new entityhealth element when an entity takes damage. If an entity takes damage when it already has an entityhealth element, then it can update the health rather than create a new row, also updating the time damage was last dealt. If you want to find out someone's health, then you only need to look and see if they have an entityhealth row, or if they have a row in deadentities table. The reason this works is that an entity has an implicit boolean hidden in the row existing in the table. For the entityhealth table, that implicit boolean was ishurt from the first function. For the deadentities table, the implicit boolean of isdead, also implies a health value of 0, which can reduce processing for many other systems. If you don't have to load a float and check that it is less than 0, then you're saving a floating point comparison or conversion to boolean.

Other similar cases include weapon reloading, oxygen levels when swimming, anything that has a value that runs out, has a maximum, or has a minimum. Even things like driving speeds of cars. If they are traffic, then they will spend most of their time driving at *traffic speed* not some speed that they need to calculate. If you have a group of people all heading in the same direction, then someone joining the group can be *intercepting* until they manage to, at which point they can give up their self and become controlled by the group.

As an aside, we use a map and a list here for simplicity of reading, and though they are not the most optimal containers in that they are trees, or pointers to elements, rather than a nice contiguous array of some sort, they're not going to adversely affect the profile as much as the problem being overcome here, namely the mixing of hot[1] data into the middle of the entity class.

Another example is with AI. If you have all your entities maintain a team index, then you have to check each entity before reacting to them. If you want to avoid all the entities from team x, and head towards the closest member of team y, then if each team is in a different table you can just operate on all the entities in those tables. If you want to produce an avoidance vector from team x, you create a mapreduce function that maps team x members to an avoidance vector, then reduce by summing.

```
vector mapavoidance( entity *e ) {
  vector difference = e->pos - currentpos;
  float oneoversqr = 1.0f / difference.getsquaredlength();
  return difference * oneoversqr;
}
vector reduceavoidance( const vector l, const vector r ) {
  return l + r;
}
pair<float,entity*> mapnearest( entity *e ) {
  vector difference = e->pos - currentpos;
  float squaredistance = difference.getsquaredlength();
  return pair<float,entity*>( squaredistance, e );
}
pair<float,entity*> reducenearest( pair<float,entity*> l, pair<
    float,entity*> r ) {
  if( l.left < r.left ) return l; return r;
}
```

Listing 2.5: map and reduce

The implicit boolean in these tables is that they are worth avoiding, and that they are worth aiming towards. If nothing maps, then the seeding value to the reduce is returned.

Another use is in state management. If an AI hears gunfire, then they can add a row to a table for when they last heard gunfire, and

[1] hot data is the member or members of a class that are accessed with high frequency, and there can even be multiple per class.

that can be used to determine whether they are in a heightened state of awareness. If an AI has been involved in a transaction with the player, it is important that they remember what has happened as long as the player is likely to remember it. If the player has just sold an AI their +5 longsword, it's very important that the shopkeeper AI still have it in stock if the player just pops out of the shop for a moment. Some games don't even keep inventory between transactions, and that can become a sore point if they accidentally sell something they need and then save their progress.

The general concept of tacking on data, or patching loaded data with dynamic additional attributes, has been around for quite a while. Save games often encode the state of a dynamic world as a delta from the base state, and one of the first major uses was in fully dynamic environments, where a world is loaded, but can be destroyed or altered afterwards. Some world generators took a procedural landscape and allowed their content creators to add patches of extra information, villages, forts, outposts, or even break out landscaping tools to drastically adjust the generated data. Taking that patching and applying it to in-game runtime information adds a level of control not normally available without using intrusive techniques.

2.3 Don't use enums

Enumerations are used to define sets of states. We could have had a state variable for the regenerating entity, one that had infullhealth, ishurt, isdead as its three states. We could have had a team index variable for the avoidance entity enumerating all the available teams. Instead we used tables to provide all the information we needed. Any enum can be emulated with a variety of tables. All you need is one table per enumerable value. Setting the enumeration is an insert into a table.

When using tables to replace enums, some things become more difficult: finding out the value of an enum in an entity is difficult as it requires checking all the tables that represent that state for the

entity. However, this is mostly disallowed and unnecessary, as the main reason for getting the value is either to do an operation based on the state, or to find out if an entity is in the right state to be considered for an operation.

If the enum is a state or type enum previously handled by a switch or virtual call, then we don't need to look up the value, instead we change the way we think about the problem. The solution is to run transforms taking the content of each of the switch cases or virtual methods as the operation to apply to the appropriate table, the table corresponding to the original enumeration value.

If the enum is instead used to determine whether or not an entity can be operated upon, then you can operate on all the entities in the appropriate table. Transforming the whole table or the join of that table with some other condition. If you're thinking about the case where you have an entity as the result of a query and need to know if it is in a certain state before deciding commit some change, consider that the table you need access to could have been one of the initial mappings, meaning that no entity would be found that didn't match the enum replacing table in the way you needed it to.

2.4 Prelude to polymorphism

Let's consider now how we implement polymorphism. We know we don't have to use a virtual table pointer; we could use an enum as a type variable. That variable, the member of the structure that defines at runtime what that structure should be capable of and how it is meant to react. That variable could be used to direct the choice of functions called when methods are called on the object.

When your type is defined by a member type variable, it's usual to implement virtual functions as switches based on that type, or as an array of functions. If we want to allow for runtime loaded libraries, then we would need a system to update which functions are called. The humble switch is unable to accommodate this, but the array of functions could be modified at runtime.

We have a solution, but it's not elegant, or efficient. The data is still in charge of the instructions, and we suffer the same instruction cache hit whenever a virtual function is unexpected. However, when we don't really use enums, but instead tables that represent each possible value of an enum, it is still possible to keep compatible with dynamic library loading the same as with pointer based polymorphism, but we also gain the efficiency of a data-flow processing approach to processing heterogeneous types.

For each class, instead of a class declaration, we have a factory that produces the correct selection of table insert calls. Instead of a polymorphic method call, we utilise existence based processing. Our elements in a tables allow the characteristics of the class to be implicit. Creating your classes with factories can easily be extended by runtime loaded libraries. Registering a new factory should be simple as long as there is a data driven factory method. The processing tables and their `update()` functions would also be added to the main loop.

2.5 Dynamic runtime polymorphism

If you create your classes by composition, and you allow the state to change by inserting and removing from tables, then you also allow yourself access to dynamic runtime polymorphism.

Polymorphism is the ability for an instance in a program to react to a common entry point in different ways due only to the nature of the instance. In C++, compile time polymorphism can be implemented through templates and overloading. Runtime polymorphism is the ability for a class to provide a different implementation for a common base operation with the class type unknown at compile time. C++ handles this through virtual tables, calling the right function at runtime based on the type hidden in the virtual table pointer at the start of the memory pointed to by the this pointer. Dynamic runtime polymorphism is when a class can react differently to a common call signature in different ways based on both it's type

and any other internal state. C++ doesn't implement this explicitly, but if a class allows the use of an internal state variable or variables, it can provide differing reactions based on that state as well as the core language runtime virtual table lookup. Consider the following code:

```
class shape {
public:
  shape() {}
  virtual ~shape() {}
  virtual float getarea() const = 0;
};
class circle : public shape {
public:
  circle( float diameter ) : d(diameter ) {}
  ~circle() {}
  float getarea() const { return d*d*pi/4; }
  float d;
};
class square : public shape {
public:
  square( float across ) : width( across ) {}
  ~square() {}
  float getarea() const { return width*width; }
  float width;
};
void test() {
  circle circle( 2.5f );
  square square( 5.0f );
  shape *shape1 = &circle, *shape2 = &square;
  printf( "areas are %f and %f\n", shape1->getarea(), shape2->
      getarea() );
}
```

Listing 2.6: simple object-oriented shape code

Allowing the objects to change shape during their lifetime requires some compromise in C++ one way is to keep a type variable inside the class.

```
enum shapetype { circletype, squaretype };
class mutableshape {
public:
  mutableshape( shapetype type, float argument )
    : m_type( type ), distanceacross( argument )
    {}
  ~mutableshape() {}
  float getarea() const {
```

```
    switch( m_type ) {
      case circletype: return distanceacross*distanceacross*pi/4;
      case squaretype: return distanceacross*distanceacross;
    }
  }
  void setnewtype( shapetype type ) {
    m_type = type;
  }
  shapetype m_type;
  float distanceacross;
};
void testinternaltype() {
  mutableshape shape1( circletype, 5.0f );
  mutableshape shape2( circletype, 5.0f );
  shape2.setnewtype( squaretype );
  printf( "areas are %f and %f\n", shape1.getarea(), shape2.
      getarea() );
}
```

Listing 2.7: ugly internal type code

A better way is to have a conversion function to handle each case.

```
square squarethecircle( const circle &circle ) {
  return square( circle.d );
}
void testconvertintype() {
  circle circle( 5.0f );
  square square = squarethecircle( circle );
}
```

Listing 2.8: convert existing class to new class

Though this works, all the pointers to the old class are now invalid. Using handles would mitigate these worries, but add another layer of indirection in most cases, dragging down performance even more.

If you use existence-based-processing techniques, your classes defined by the tables they belong to, then you can switch between tables at runtime. This allows you to change behaviour without any tricks, without any overhead of a union to carry all the differing data around for all the states that you need. If you compose your class from different attributes and abilities then need to change them post creation, you can. Looking at it from a hardware point of view, in

order to implement this form of polymorphism you need a little extra space for the reference to the entity in each of the class attributes or abilities, but you don't need a virtual table pointer to find which function to call. You can run through all entities of the same type increasing cache effectiveness, even though it provides a safe way to change type at runtime.

As another potential benefit, the implicit nature of having classes defined by the tables they belong to, there is an opportunity to register a single entity with more than one table. This means that not only can a class be dynamically runtime polymorphic, but it can also be multimorphic in the sense that it can be more than one class at a time. A single entity might react in two different ways to the same trigger call because that might be appropriate for that current state for that class. This kind of multidimensional classing doesn't come up much in gameplay code, but in rendering, there are usually a few different axes of variation such as the material, what blend mode, what kind of skinning or other vertex adjustments are going to take place on a given instance. Maybe we don't see this flexibility in gameplay code because it's not available through the natural tools of the language.

2.6 Event handling

When you wanted to listen for events in a system in the old days, you'd attach yourself to an interrupt. Sometimes you might get to poke at code that still does this, but it's normally reserved for old or microcontroller scale hardware. The idea was simple, the processor wasn't really fast enough to poll all the possible sources of information and do something about the data, but it was fast enough to be told about events and process the information as and when it arrived. Event handling in games has often been like this, register yourself as interested in an event, then get told about it when it happens. The publish and subscribe model has been around for many years, but there's not been a standard interface built for it

as it often requires from problem domain knowledge to implement effectively. Some systems want to be told about every event in the system and decide for themselves, such as windows event handling. Some systems subscribe to very particular events but want to react to them as soon as they happen, such as handlers for the BIOS events like the keyboard interrupt.

Using your existence in a table as the registration technique makes this simpler than before and lets you register and de-register with great pace. You can register an entity as being interested in events, and choose to fire off the transform immediately, or queue up new events until the next loop round. As the model becomes simpler and more usable, the opportunity for more common use leads us to new ways of implementing code traditionally done via polling.

For example: unless the player character is within the distance to activate a door, the event handler for the player's action button wont be attached to anything door related. When the character comes within range, the character registers into the *has_pressed_action* event table with the *open_door_(X)* event result. This reduces the amount of time the CPU wastes figuring out what thing the player was trying to activate, and also helps provide state information such as on-screen displays saying that *pressing Green will Open the door*. It may even do this by hooking into low level tables generated by default such as a *character registers into the has_pressed_action event table* event.

This coding style is somewhat reminiscent of aspect oriented programming where it is easy to allow for cross-cutting concerns in the code. In aspect oriented programming, the core code for any activities is kept clean, and any side effects or vetoes of actions are handled by other concerns hooking into the activity from outside. This keeps the core code clean at the expense of not knowing what is really going to be called when you write a line of code. How using registration tables differs is in where the reactions come from and how they are determined. As we shall see in chapter 5, when you use conditions for logical reactions, debugging can become significantly simpler as the barriers between cause and effect normally implicit

in aspect oriented programming are significantly diminished or removed, and the hard to adjust nature of object oriented decision making can be softened to allow your code to become more dynamic without the normal associated cost of data driven control flow.

Chapter 3

Component Based Objects

Component oriented design is a good head start for high level data-oriented design. Components put your mind in the right frame for not linking together concepts when they don't need to be, and component based objects can easily be processed by type, not by instance, which leads to a smoother processing profile. Component based entity systems are found in games development as a way to provide a data-driven functionality system for entities, which can allow for designer control over what would normally be the realm of a programmer. Not only are component based entities better for rapid design changes, but they also stymie the chances of the components getting bogged down into monolithic objects as most game designers would demand more components with new features over extending the scope of existing components. Most new designs need iterating, and extending an existing component by code doesn't allow design to change back and forth, trying different things.

When people talk about how to create compound objects, sometimes they talk about using components to make up an object.

Though this is better than a monolithic class, it is not component based approach, it merely uses components to make the object more readable, and potentially more reusable. Component based turns the idea of how you define an object on its head. The normal approach to defining an object in object oriented design is to name it, then fill out the details as and when they become necessary. For example, your car object is defined as a Car, if not extending Vehicle, then at least including some data about what physics and meshes are needed, with construction arguments for wheels and body shell model assets etc, possibly changing class dependent on whether it's an AI or player car. In component oriented design, component based objects aren't so rigidly defined, and don't so much become defined after they are named, as much as a definition is selected or compiled, and then tagged with a name if necessary. For example, instancing a physics component with four wheel physics, instancing a renderable for each part (wheels, shell, suspension) adding an AI or player component to control the inputs for the physics component, all adds up to something which we can tag as a Car, or leave as is and it becomes something implicit rather than explicit and immutable.

A component based object is nothing more than the sum of its parts. This means that the definition of a component based object is also nothing more than an inventory with some construction arguments. This object or definition agnostic approach makes refactoring and redesigning a completely trivial exercise.

3.1 Components in the wild

Component based approaches to development have been tried and tested. Many high profile studios have used component driven entity systems to great success[1], and this was in part due to their developer's unspoken understanding that objects aren't a good place to store all your data and traits. Gas Powered Games' Dungeon Siege

[1]Gas Powered Games, Looking Glass Studios, Insomniac, Neversoft all used component based objects

Architecture is probably the earliest published document about a game company using a component based approach. If you get a chance, you should read the article[2]. In the article it explains that using components means that the entity type[3] doesn't need to have the ability to do anything. Instead, all the attributes and functionality come from the components of which the entity is made.

In this section we'll show how we can take an existing class and rewrite it in a component based fashion. We're going to tackle a fairly typical complex object, the player class. Normally these classes get messy and out of hand quite quickly, and we're going to assume it's a player class designed for a generic 3rd person action game, and take a typically messy class as our starting point.

```
1  class Player {
2  public:
3    Player();
4    ~Player();
5    Vec GetPos(); // the root node position
6    void SetPos( Vec ); // for spawning
7    Vec GetSpeed(); // current velocity
8    float GetHealth();
9    bool IsDead();
10   int GetPadIndex(); // the player pad controlling me
11   float GetAngle(); // the direction the player is pointing
12   void SetAnimGoal( ... ); // push state to anim-tree
13   void Shoot( Vec target ); // fire the player's weapon
14   void TakeDamage( ... ); // take some health off, maybe animate
         for the damage reaction
15   void Speak( ... ); // cause the player to start audio/anim
16   void SetControllable( bool ); // no control in cut-scene
17   void SetVisible( bool ); // hide when loading / streaming
18   void SetModel( ... ); // init streaming the meshes etc
19   bool IsReadyForRender();
20   void Render(); // put this in the render queue
21   bool IsControllable(); // player can move about?
22   bool IsAiming(); // in normal move-mode, or aim-mode
23   bool IsClimbing();
24   bool InWater(); // if the root bone is underwater
25   bool IsFalling();
26   void SetBulletCount( int ); // reload is -1
27   void AddItem( ... ); // inventory items
```

[2]Gas Powered Games released a web article about their component based architecture back in 2004, read it at http://garage.gaspowered.com/?q=su_301

[3]GPG:DG uses GO or Game-Objects, but we stick with the term entity because it has become the standard term.

```
28    void UseItem( ... );
29    bool HaveItem( ... );
30    void AddXP( int ); // not really XP, but used to indicate when
          we let the player power-up
31    int GetLevel(); // not really level, power-up count
32    int GetNumPowerups(); // how many we've used
33    float GetPlayerSpeed(); // how fast the player can go
34    float GetJumpHeight();
35    float GetStrength(); // for melee attacks and climb speed
36    float GetDodge(); // avoiding bullets
37    bool IsInBounds( Bound ); // in trigger zone?
38    void SetGodMode( bool ); // cheater
39  private:
40    Vec pos, up, forward, right;
41    Vec velocity;
42    Array<ItemType> inventory;
43    float health;
44    int controller;
45    AnimID currentAnimGoal;
46    AnimID currentAnim;
47    int bulletCount;
48    SoundHandle playingSoundHandle; // null most of the time
49    bool controllable;
50    bool visible;
51    AssetID playerModel;
52    LocomotionType currentLocomotiveModel;
53    int xp;
54    int usedPowerups;
55    int SPEED, JUMP, STRENGTH, DODGE;
56    bool cheating;
57  };
```

Listing 3.1: Player class

3.2 Away from the hierarchy

A recurring theme in articles and post-mortems from people moving from Object-Oriented hierarchies of gameplay classes to a component based approach is the transitional states of turning their classes into containers of smaller objects, an approach often called composition. This transitional form takes an existing class and finds the boundaries between the concepts internal to the class and refactors them out into new classes that can be pointed to by the original class.

First, we move the data out into separate structures so they can be more easily combined into new classes.

```
struct PlayerPhysical {
  Vec pos, up, forward, right;
  Vec velocity;
};
struct PlayerGameplay {
  float health;
  int xp;
  int usedPowerups;
  int SPEED, JUMP, STRENGTH, DODGE;
  bool cheating;
};
struct EntityAnim {
  AnimID currentAnimGoal;
  AnimID currentAnim;
  SoundHandle playingSoundHandle; // null most of the time
};
struct PlayerControl {
  int controller;
  bool controllable;
};
struct EntityRender {
  bool visible;
  AssetID playerModel;
};
struct EntityInWorld {
  LocomotionType currentLocomotiveModel;
};
struct Inventory {
  Array<ItemType> inventory;
  int bulletCount;
};

SparseArray<PlayerPhysical> phsyicalArray;
SparseArray<PlayerGameplay> gameplayArray;
SparseArray<EntityAnim> animArray;
SparseArray<PlayerControl> controlArray;
SparseArray<EntityRender> renderArray;
SparseArray<EntityInWorld> inWorldArray;
SparseArray<Inventory> inventoryArray;

class Player {
public:
  Player();
  ~Player();
  // ...
  // ... the member functions
  // ...
private:
```

```
49    int EntityID;
50  };
```

When we do this, we see how a first pass of building a class out of smaller classes can help organise the data into distinct purpose oriented collections, but we can also see the reason why a class ends up being a tangled mess. The rendering functions need access to the player's position as well as the model, and the gameplay functions such as Shoot need access to the inventory as well as setting animations and dealing damage. Take damage will need access to the animations, health. Things are already seeming more difficult to handle than expected. But what's really happening here is that you can see that code needs to cut across different pieces of data. With this first pass, we can start to see that functionality and data don't belong together.

We move away from the class being a container for the classes immediately as we can quickly see the benefit of cache locality when we're iterating over multiple entities doing related tasks. If we were iterating over the entities checking for whether they were out of bullets, and if so setting their animation to reloading, we would only need to check the EntityID to find the element for each involved entity.

Functionality of a class, or an object, comes from how facts are manipulated. The relations between the facts are part of the problem domain, but the facts are only raw data. This separation of fact from meaning is not possible with an object oriented approach, which is why every time a fact acquires a new meaning, the meaning has to be implemented as part of the class containing the fact. Removing the facts from the class and instead keeping them as separate components has given us the chance to move away from classes that have meaning, but at the expense of having to look up facts from different sources.

3.3 Towards Managers

After splitting your classes up into components, you might find that your classes look more awkward now they are accessing variables hidden away in new structures. But it's not your classes that should be looking up variables, but instead transforms on the classes. A common operation such as rendering requires the position and the model information, but it also requires access to the renderer. Such global access is seen as a common compromise during most game development, but here it can be seen as the method by which we move away from a class centric approach to transforming our data into render requests that affect the graphics pipeline without referring to data unimportant to the renderer, and how we don't need a controller to fire a shot.

```
class RenderManager {
  void Update() {
    foreach( {index,pos} in positionArray ) {
      if( index in renderArray ) {
        ModelID mid = renderArray[ index ].playerModel;
        gRenderer.AddModel( mid, pos );
      }
    }
  }
}
class PhysicsManager {
  void Update() {
    foreach( {index,pos} in positionArray ) {
      if( index in renderArray ) {
        ModelID mid = renderArray[ index ].playerModel;
        ApplyCollisionAndResponse( pos.position, pos.velocity
      }
    }
  }
}
class PlayerControl {
  void Update() {
    foreach( {index,control} in controlArray ) {
      if( control.controllable ) {
        Pad pad = GetPad( control.controller );
        if( pad.IsPressed( SHOOT ) ) {
          if( index in inventoryArray ) {
            if( inventoryArray[ index ].bulletCount > 0 ) {
              if( index in animArray ) {
                animArray[ index ].currentAnimGoal = SHOOT_ONCE;
              }
```

```
32              }
33            }
34          }
35        }
36      }
37      foreach( {index,inv} in inventoryArray )  {
38        if( inv.bulletCount > 0 ) {
39          if( index in animArray ) {
40            anim = animArray[ index ];
41            if( anim.currentAnim == SHOOT_ONCE ) {
42              anim.currentAnim = SHOT;
43              inventoryArray[index].bulletCount -= 1;
44              anim.playingSoundHandle = PlaySound( GUNFIRE );
45            }
46          }
47        }
48      }
49    }
50  }
```

What happens when we let more than just the player use these arrays? Normally we'd have some separate logic for handling player fire until we refactored the weapons to be generic weapons with NPCs using the same code for weapons proably by having a new weapon class that can be pointed to by the player or an NPC, but instead what we have here is a way to split off the weapon firing code in such a way as to allow the player and the NPC to share firing code without inventing a new class to hold the firing. In fact, what we've done is split the firing up into the different tasks that it really contains.

Tasks are good for parallel processing, and with component based objects we open up the opportunity to make most of our previously class oriented processes into more generic tasks that can be dished out to whatever CPU or co-processor can handle them.

3.4 There is no Entity

What happens when we completely remove the player class? If we consider that an entity may be not only represented by it's collection

of components, but also might only be it's current configuration of components, then there is the possibility of removing the core class of Player. Removing this class can mean we no longer think of the player as being the centre of the game, but also the class no longer existing means that any code is no longer tied to itself.

```
struct Orientation { Vec pos, up, forward, right; };
SparseArray<Orientation> orientationArray;
SparseArray<Vec> velocityArray;
SparseArray<float> healthArray;
SparseArray<int> xpArray, usedPowerupsArray, controllerID,
    bulletCount;
struct Attributes { int SPEED, JUMP, STRENGTH, DODGE; };
SparseArray<Attributes> attributeArray;
SparseArray<bool> godmodeArray, controllable, isVisible;
SparseArray<AnimID> currentAnim, animGoal;
SparseArray<SoundHandle> playingSound;
SparseArray<AssetID> modelArray;
SparseArray<LocomotionType> locoModelArray;
SparseArray<Array<ItemType> > inventoryArray;

void RenderUpdate() {
  foreach( {index,assetID} in modelArray ) {
    if( index in isVisible ) {
      gRenderer.AddModel( assetID, orientationArray[ index ] );
    }
  }
}
void PhysicsUpdate {
  foreach( {index,vel} in velocityArray ) {
    if( index in modelArray ) {
      ApplyCollisionAndResponse( orientationArray[index], vel,
          modelArray[ index ] );
    }
  }
}
void ControlUpdate() {
  foreach( {index,control} in controllerID ) {
    if( controllable[ index ] ) {
      Pad pad = GetPad( control.controller );
      if( pad.IsPressed( SHOOT ) ) {
        if( inventoryArray[ index ].bulletCount > 0 ) {
          animArray[ index ].currentAnimGoal = SHOOT_ONCE;
        }
      }
    }
  }
}
void UpdateAnims() {
  foreach( {index, anim} in currentAnim ) {
```

```
43      if( anim == SHOOT_ONCE ) {
44        anim = SHOT;
45        inventoryArray[index].bulletCount -= 1;
46        playingSound = PlaySound( GUNFIRE );
47      }
48    }
49  }
50  int NewPlayer( int padID, Vec startPoint ) {
51    int ID = newID();
52    controllerID[ ID ] padID;
53    GetAsset( "PlayerModel", ID ); // adds a request to put the
          player model into modelArray[ID]
54    orientationArray[ ID ] = Orientation(startPoint);
55    velocityArray[ ID ] = VecZero();
56    return ID;
57  }
```

Moving away from a player class means that many other classes can be invented without adding much code. Allowing script to generate new classes by composition increases the power of script to dramatically increase the apparent complexity of the game without adding more code. What is also of major benefit is that all the different entities in the game now run the same code at the same time. everyone's physics is updated before they are rendered[4] and everyone's controls (whether they be player or AI) are updated before they are animated. Having the managers control when the code is executed is a large part of the leap towards fully parallelisable code.

[4]or is updated while the rendering is happening on another thread

Chapter 4

Hierarchical Level-of-Detail and Implicit-state

Consoles and graphics cards are not generally bottlenecked at the polygon rendering stage in the pipeline. Usually they are fill rate bound if there are large polygons on screen, or there is a lot of alpha blending, and for the most part, graphics chips spend a lot of their time reading textures. Because of this, the old way of doing level of detail with multiple meshes with decreasing numbers of polygons is never going to be as good as a technique that takes into account the actual data required of the level of detail used in each renderable. Hierarchical level of detail fixes the problem of high primitive count and mistargetted art optimisations by grouping and merging many low level of detail meshes into one low level of detail mesh, thus reducing the time spent in the setup of render calls, and enforcing a better perspective on the artist producing the lower resolution asset. In a typical very large scale environment, a hierarchical level of detail implementation can reduce the workload on a game engine by

an order of magnitude as the number of entities in the scene considered for rendering drops significantly. Even though the number of polygons rendered might be exactly the same, or maybe even more, the fact that the engine usually only has to handle a static number of entities at once increases stability and allows for more accurately targeted optimisations of both art and code.

4.1 Existence from Null to Infinity

Going back to our entities being implicit on their attributes, we can utilise the theory of hierarchical level of detail to offer up some optimisations for our code. If we have high level low fidelity coarse grain game logic for distant or currently unimportant game entities, and only drill down to highly tuned code when an entity becomes more apparent to the player, then we can save a lot of cycles on things the player is not interested in, or possibly even able to see.

Consider a game where you are defending a base from incoming attacks. The attackers come in squadons of ships, you can see them all coming at once, over a thousand ships in all and up to twenty at once in each squadron. You have to shoot them down, or be inundated with gunfire and bombs, taking out both you and the base you are defending.

Running full AI, with swarming for motion and avoidance for your slower moving weapons might be too much if it was run on all thousand ships every tick, but you don't need to. The basic assumption made by most AI programmers is that unless they are within attacking range, then they don't need to be running AI. This is true, and does offer an immediate speed up compared to the naive approach. Hierarchical LOD provides another way to think about this, through changing the number of entities based on how they are perceived by the player. For want of a better term, count-lodding is a name that describes what is happening behind the scenes a little better, because sometimes there is no hierarchy and yet there can still be a change in count between the levels of detail.

In the count-lodding version of the base defender game, there is a few Wave entities that project a few Squadron blips on the radar. The squadrons don't exist as their own entities until they get close enough. Once a wave's squadron is within range, the Wave can decrease its squadron count, and pop out a new Squadron entity. The Squadron entity shows blips on the radar for each of its component ships. The ships don't exist, but they are implicit in the Squadron. The Wave continues to pop Squadrons as they come into range, and once it's internal count has dropped to zero, it deletes itself as it now represents no entities. As a Squadron comes into even closer range, it pops out its ships into their own entities, and deletes itself. As the ships get closer, their renderables are allowed to switch to higher resolution and their AI is allowed to run at a higher intelligence setting.

When the ships are shot at, they switch to a taken damage type much like the health system earlier. They are full health unless they take damage. If an AI reacts to damage with fear, they may eject, adding another entity to the world. If the wing of the plane is shot off, then that also becomes a new entity in the world. Once a plane has crashed, it can delete its entity and replace it with a smoking wreck entity that will be much simpler to process than an aerodynamic simulation.

If the player can't keep the ships at bay and their numbers increase in size so much that any normal level of detailing can't kick in, count lodding can still help by returning ships to squadrons and flying them around the base attacking as a group rather than as invididual ships. The level of detail heuristic can be tuned so that the nearest and front-most squadron are always the highest level of detail, and the ones behind the player maintain a very simplistic representation.

This is game development smoke and mirrors as a basic game engine element. In the past we have reduced the number of concurrent attacking AI[1], reduced the number of cars on screen by staggering

[1] I beleive this was half-life

the lineup over the whole race track[2], and we've literally combined people together into one person instead of having loads of people on screen at once[3]. This kind of reduction of processing is commonplace. Now use it everywhere appropriate, not just when a player is not looking.

4.2 Mementos

Reducing detail introduces an old problme, though. Changing level of detail in game logic systems, AI and such, brings with it the loss of high detail history. In this case we need a way to store what is needed to maintain a highly cohesive player experience. If a high detail squadron in front of the player goes out of sight and another squadron takes their place, we still want any damage done to the first group to reappear when they come into sight again. Imagine if you had shot out the glass on all the ships and when they came round again, it was all back the way it was when they first arrived. A cosmetic effect, but one that is jarring and makes it harder to suspend disbelief.

When a high detail entity drops to a lower level of detail, it should store a memento, a small, well compressed nugget of data that contains all the necessary information in order to rebuild the higher detail entity from the lower detail one. When the squadron drops out of sight, it stores a memento containing compressed information about the amount of damage, where it was damaged, and rough positions of all the ships in the squadron. When the squadron comes into view once more, it can read this data and generate the high detail entities back in the state they were before. Lossy compression is fine for most things, it doesn't matter precisely which windows, or how they were cracked, maybe just that about 2/3 of the windows were broken.

[2]Ridge Racer was one of the worst for this
[3]Populous did this

Another good example is in a city based free-roaming game. If AIs are allowed to enter vehicles and get out of them, then there is a good possibility that you can reduce processing time by removing the AIs from world when they enter a vehicle. If they are a passenger, then they only need enough information to rebuild them and nothing else. If they are the driver, then you might want to create a new driver type based on some attributes of the pedestrian before making the memento for when they exit the vehicle.

If a vehicle reaches a certain distance away from the player, then you can delete it. To keep performance high, you can change the priorities of vehicles that have mementos so that they try to lose sight of the player thus allowing for earlier removal from the game. Optimisations like this are hard to coordinate in Object-oriented systems as internal inspection of types isn't encouraged.

4.3 JIT Data Generation

If a vehicle that has been created as part of the ambient population is suddenly required to take on a more important role, such as the car being involved in a fire fight. It is important to generate new entities that don't seem overly generic or unlikely given what the player knows about the game so far. Generating data can be thought of as providing a memento to read from just in time. JIT mementos are faked mementos that provide a false sense of continuity by allowing pseudo random generators or hash functions the opportunity to replace non-zero memory usage mementos when the data is unlikely to change, or be needed for more than a short while.

JIT mementos can also be the basis of a highly textured environment with memento style sheets or style guides that can direct a feel bias for any mementos generated in those virtual spaces. Imagine a city style guide that specifies rules for occupants of cars, that businessmen might share, but are less likely to, families have children in the back seats with mum or dad driving, young adults tend to drive around in pairs. These style guides help add believability

to any generated data. Add in local changes such as having types of car linked to their drivers, convertibles driven by well dressed types or kids, low riders driven almost exclusively by ghetto residents, imports driven by young adults. In a space game, dirty hairy pilots of cargo ships, well turned out officers commanding yachts, rough and ready mercenaries in everything from a single seater to a dreadnought.

JIT mementos are a good way to keep variety up, and style guides bias that so it comes without the impression that everyone is different so everyone is the same. When these biases are played out without being striclty adhered to, you can build a more textured environment. If your environment is heavily populated with a completely different people all the time, there is nothing to hold onto, not patterns to recognise. When there are no patterns, the mind tends to see noise, or mark it as samey. Even the most varied virtual worlds look bland when there is too much content all in the same place.

4.4 Alternative Axes

As with all things, take away an assumption and you can find other uses for a tool. Whenever you read about or work with a level of detail system, you will be aware that the constraint on what level of detail is shown has always been some distance function in space. It's now time to take that assumption, discard it, and analyse what is really happening.

First, we find that if we take away the assumption of distance, we can infer the conditional as some kind of linear measure. This value normally comes from a function that takes the camera position and finds the relative distance to the entity under consideration. What we may also realise when discarding the distance assumption is a more fundamental understanding of that what we are trying to do. We are using a runtime variable to control the presentation state of an entity. We use runtime variables to control the state of

many parts of our game already, but in this case, there is a passive presentation response to the variable, or axis being monitored. The presentation is usually some graphical, or logical level of detail, but it could be something as important to the entity as its own existence.

How long until a player forgets about something that might otherwise be important? This information can help reduce memory usage as much as distance. If you have ever played *Grand Theft Auto IV*, you might have noticed that the cars can dissappear just by not looking at them. As you turn around a few times you might notice that the cars seem to be different each time you face their way. This is a stunning use of temporal level of detail. Cars that have been bumped into or driven and parked by the player remain where they were, because, in essence, the player put them there. Because the player has interacted with them, they are likely to remember that they are there. However, ambient vehicles, whether they are police cruisers, or civilian vehicles, are less important and don't normally get to keep any special status so can vanish when the player looks away.

In adition to time-since-seen, some elements may base their level of detail on how far a player has progressed in the game, or how many of something a player has, or how many times they have done it. For example, a typical bartering animation might be cut shorter and shorter as the game uses the axis of *how many recent barters* to draw back the length of any non-interactive sections that could be caused by the event. This can be done simply, and the player will be thankful. It may even be possible to allow for multi-item transactions after a certain number of transactions have happened. In effect, you could set up gameplay elements, reactions to situations, triggers for tutorials or extensions to gameplay options all through these abstracted level of detail style axes.

This way of manipulating the present state of the game is safer from transition errors. Errors that happen because going from one state to another may have set something to true when transitioning one direction, but not back to false when transitioning the other way. You can think of the states as being implicit on the axis, not explicit,

calculated purely as a triggered event that manipulates state.

An example of where transition errors occur is in menu systems where though all transitions should be reversible, sometimes you may find that going down two levels of menu, but back only one level, takes you back to where you started. For example, entering the options menu, then entering an adjust volume slider, but backing out of the slider might take you out of the options menu all together. These bugs are common in UI code as there are large numbers of different layers of interaction. Player input is often captured in obscure ways compared to gameplay input response. A common problem with menus is one of ownership of the input for a particular frame. For example, if a player hits both the forward and backward button at the same time, a state machine UI might choose to enter whichever transition response comes first. Another might manage to accept the forward event, only to have the next menu accept the back event, but worst of all might be the unlikely but seen in the wild, menu transitioning to two different menus at the same time. Sometimes the menu may transition due to external forces, and if there is player input captured in a different thread of execution, the game state can become disjoint and unresponsive. Consider a network game's lobby, where if everyone is ready to play, but the host of the game disconnects while you are entering into the options screen prior to game launch, in a traditional state machine like approach to menus, where should the player return to once they exit the options screen? The lobby would normally have dropped you back to a server search screen, but in this case, the lobby has gone away to be replaced with nothing. This is where having simple axes instead of state machines can prove to be simpler to the point of being less buggy and more responsive.

Chapter 5

Condition Tables

Condition tables are a data-oriented approach to Decision tables. Decision tables are a method by which you can generate flow control with a table, mapping a vector of predicates to a set of outcomes or destinations. In decision tables, each row has a predicate – a logical statement, such as isTrue, isFalse, null, or some boolean returning function on the data present in the column, such as > 0 for numbers, or `contains( "win" )` for strings. Decision tables aren't entirely like standard flow control statements, as they can lead to multiple outcomes. We will see that a condition table query can also end up being true for multiple outcomes, or even have no matching rows.

5.1 Building Questions

Condition tables are similar to Decision tables, except they have some restrictions that allow for easier optimisation. All the input columns in a condition table query are boolean. If you are familiar with decision tables, this might seem like a limiting constraint, however if you need to check that $X < 20$, then you can do that before it enters the query. Also, unlike decision tables, each row in

a condition table doesn't have a function to call, but instead a table reference to output to if there is a match on the conditions. These simple constraints allow for existence-based-processing of the result tables, thus reducing the instruction cache misses of the more traditional database oriented decision table approach, and also allows the driving of event tables so that conditions can trigger events on subscribed entities.

For each component involved in the condition table operation, you generate the boolean values into the condition table input. Generating each boolean value from the data in this fashion can utilise the struct-of-arrays format much better, as the predicate can run as a tiny kernel over the streaming data. In a traditional array-of-structs format, this technique wouldn't improve performance or increase cache-line utilisation, so unless you're testing your code on well oriented data, this whole preprocessing step might be worse than pointless. If your data is organised by concept, such as in an object oriented class, then you might end up loading in the same data multiple times to prepare the input table. As each kernel may be accessing only a few bytes of a structure at a time, this will waste memory bandwidth.

Once the condition input is prepared, we then prepare the transform. The transform consists of a set of query arrays and an output. The output will be multiple tables, each query array can point to any of the output tables, which means that an entry can end up in more than one table, and can be put into a table more than one time based on how the output table is organised. Sometimes, an entity will want to only act on the first matching decision point (this would be a way to implement behaviour trees), sometimes a component will want to include multiple matches (this would be a way to implement scoring points).

The condition table is made up of rows, each row has a query array that consists of one or more tri-state columns with possible values of { isTrue, isFalse, null }. To match with an input, all query array columns must match. In effect, an *and* over all the element queries. The null entry will always return true. With this, it is

possible to build up any query you would normally do on your data.

5.2 Flip it on its head

The decision table approach is another form of taking what is normally a singular entity approach and flipping it on its head. Now we are aware of the idea of a structure of arrays, we can start to see other areas of code that previously were being driven by a multiple individual entities entering into update functions and calling things based on their own data, and causing the processor to change direction and try to keep up with the winding trail set by all the selfish entities. If every entity needs to do an update, then effectively no individual entity needs to do an update. Components need to update, and if instead of running an update over each and every entity in your application, you run a component update over every component manager, then you're moving from a data driven mess to a stream processing or flow based programming paradigm. Each of these approaches has benefits, but the difficulty in using them in the past has been how to begin to do it when starting with an object oriented data layout. If your data is already organised as structures of arrays (or indeed component managers of arrays of transform related data), then the flow based techniques for transforming data don't need to be shoe-horned in.

Because we have moved the decisions out of the instances, we are in a position to do more at the same time. A wary object-oriented programmer might assume that each decision table applies to only one set of entities, or one set of queries, but the truth is far more parallel. Once you have moved away from entities being a type, and instead let them be implicit on their components, you find that you can process a very large number of apparently disparate entities at the same time. As you build your application's update function, you find that you need to make decisions for components, update components based on those decisions, and sometimes iterate over that sequence a few times. At no point are you limited to one entity

at a time, or even one component at a time unless you build in dependencies into your transform chain.

Moving away from single entity processing is also good if you want to offload your core application code onto a compute shader. As long as you maintain your data in simple arrays inside the components, then there will likely be very little currying of data into a shader ready format. If you make sure your processing is stateless, as in it does not accumulate or modify global state, or reference other elements in the stream in a random access fashion, then it's very likely you will be able to convert your code to a compute kernel. At this point, you will have left the general purpose CPU to scheduling the operations, and joining tables. These are the only jobs still not easily handled by a stateless streaming approach.

5.3 Logging decisions for debug: Making answers talk for themselves

With all this raw data moving around, some may think that it would be harder to debug than a human friendly object-oriented code layout. Indeed, some developers think that you need to understand the data in a problem centric way in order to figure out what is causing a bug, however, normally a bug is where the implementation did what was written, not what was meant, and that can usually be found through understanding what is going on inside the software, and not the problem that was being solved by it. Most bug slaying involves understanding what the computer was doing before it did the unexpected. It is hardly ever about understanding the design, or the concepts represented by the classes. It is very unlikely that your bug exists in something as high level as the fact that a house is not a car.

For example, if you have a bug where a specific mesh is being culled, you might think that you need to see the data about the entity that renders that mesh all in one place. You may think that otherwise you could spend a lot of time traversing data structures

in a watch window just to find out in what state an object was, that caused it to flip its visible bit to false. This human view of the data may seem important to many seasoned developers, but it's false reasoning based on how difficult it is to debug and reason about data structures inside an object-oriented program. We're used to thinking about our data as objects, but our data is not objects, it's data.

If your code is data-oriented, with checkpoints between each transform, it becomes very easy to log any changes and their reason why. It even becomes simpler to rerun transforms in order to find out what conditions caused them to change. If you are using condition tables, you can run the condition check multiple times to figure out what caused them to emit or not emit output rows. If you keep the development stream-oriented, that is, no mutable state or in-place modification, then you can even allow for rolling back a lot of your program in order to find the original cause of any bad state. We can roll back when we don't have global state mutation because any previous data implies a state. As long as you don't modify the stream, but instead generate a new one as you make your way through your updates, you can always look back and see what caused the change, or at least see where the data didn't match expectation.

If you think about things like meshes as being objects that have state, then you are going to worry about how data got into a particular state. Most object oriented developers fear that without the problem domain borders, then states may get lost in one of the myriad anonymous transforms that affect the data streaming through the system, however, this is only because Object-oriented design allows or even enforces the dangerous practice of hidden data and mutable state. As it is relatively easy to hook an event handler to any table insert, adding logging for bad state changes is very easy, and if you keep your transforms deterministic, it becomes highly repeatable. In some languages there are virtual machines that allow for complete rollback of the program. There is a JVM that maintains historic state, and Microsoft's CLR provides the ability to go back in time for some of its languages. If you program your game

with condition tables, thorough logging, and a deterministic set of transforms, you can have game state rollback in a high performance language such as C++.

With this information at hand, it can be safe to assume that for the normal practice of debugging, which is figuring out how an object got into a particularly bad state, working within a well formed data oriented code base would make debugging easier, not harder.

5.4 Branching without branching

Due to the restrictions imposed on their use, condition tables have the opportunity to be realised in branch free or heavily optimised code. Depending on the number of rows in the decision table, it might be possible to optimise the query by output, but normally the query can be optimised by input. We will see more of this choice of rows vs columns in chapter ref:example.

The first thing we do is generate a truth value from the conditions and the query array. Remember the query array is made up of isTrue, isFalse, and null values. We need to generate, from an array of booleans, a single boolean result. Logically, we can produce this by xoring the inputs with the isFalse elements, and then oring in the null entries. Once we have this final array, we only need to check that all elements are true to find the entire query to be true.

If we allow for our pre-processor to produce bitmasks of input values, then we can produce a pair of bit masks that generate the kind of result we need. First, we prepare the ignore mask, this mask contains a high bit wherever we don't care about the input value. Naturally, all null fields can be converted to 1s, and so can any bits higher than the maximum query entry. The second mask is the falsity masks, and is true if the query array tri-state is isFalse. These masks are now the xor and ignore masks.

```
1 unsigned int xorMask = 0;
2 unsigned int ignoreMask = 0;
3 for( int i = 0; i < tableColumns; ++i ) {
4   if( column[i].condition == isFalse )
```

```
    xorMask |= BIT( i );
  if( column[i].condition == null )
    ignoreMask |= BIT(i);
}
for( int i = tableColumns; i < MAX_BITS; ++i ) {
  ignoreMask |= BIT(i);
}
```

Next, prepare the input as a list of boolean values. If we have well formed data, we can assume that we have already built our inputs as bit fields, but if our data is still object oriented, or at least still arrays of structures, then we will probably do this step inline with reading each structure from the array of structures.

```
foreach( row in table ) {
  unsigned int values = 0;
  for( int i = 0; i < tableColumns; ++i ) {
    if( row[i] )
      values |= BIT( i );
  }
}
```

Once you have your inputs transformed into the array of bit fields, you can run them through the condition table. If they are an array of structures, then it's likely that caching the inputs into a separate table isn't going to make things faster, so this decision pass can be run on the bit field directly. In this snippet we run one bit field through one set of condition masks.

```
unsigned int conditionsMet = ( value ^ xorMask ) | ignoreMask;
unsigned int unmet = ~conditionsMet;
if( !unmet ) {
  addToTable( t );
}
```

The code first *XORs* the values so the false values become true for all the cases where we want the value to be false. Then we *OR* with the ignore so that any entries we don't care about become true

and don't falsely break the condition. Once we have this final value, we can be certain that it will be all ones in case of a match, and will have some zeros if it is a miss. An `if` using the binary-not of the final value will suffice to prove the system works, but we can do better.

Using the *conditionsMet* value, we can generate a final mask onto an arbitrary value.

```
1 unsigned int czero = ( cvalue + 1 ); // one more than all the ones
      is zero.
2 unsigned int cmask = ~(czero | -czero) >> 31; // only zero returns
      -1 from this.
3 ID valueIn = valueOut & cmask; // if true, return the key, else
     return zero.
```

Now, we can use this value in many ways. We can use it on the primary key, and add this key to the output table, and providing a list table that can be reduced by ignoring nulls, we have, in effect, a dead bucket. If we use it on the increment counter, then we can just add an entry to the end of the output table, but not increase the size of the output table. Either way, we can assume this is a branchless implementation of an entry being added to a table. If this was an event, then a multi-variable condition check is now branchlessly putting out an event. If you let these loops unroll, then depending on your platform, you are going to process a lot more data than a branching model will.

Chapter 6

Finite State Machines

Finite state machines are a solid solution to many game logic problems. They are the building block of most medium level AI code, and provide testable frameworks for user interfaces. Notoriously convoluted finite state machines rule the world of memory card operations and network lobbies, while simple ones hold domain over point and click adventure puzzle logic. Finite state machines are highly data driven in that they react to an alphabet of signals, and change state based on both the signal received, and their current state.

6.1 Tables as States

If we wish to implement finite state machines without objects to contain them, and without state variables to instruct their flow, we can call on the runtime dynamic polymorphism inherent in the data oriented approach to provide these characteristics based only on the existence of rows in tables. We do not need an instruction cache trashing state variable that would divert the flow according to internal state. We don't need an object to contain any state dependent data for analysis of the alphabet.

At the fundamental level, a finite state machine requires a reaction to an alphabet. If we replace the alphabet handling code with condition tables, then we can produce output tables of transitions to commit. If each state is represented by a condition table, and any entity in that state represented by a row in a state table, then we can run each state in turn, collecting any transitions in buffers, then committing these changes after everything has finished a single update step.

Finite state machines can be difficult to debug due to their data-driven nature, so being able to move to a easier to debug framework should reduce development time. In my experience, being able to log all the transitions over time has reduced some AI problems down to merely looking through some logs before fixing an errant condition based only on an unexpected transion logged with its cause data.

Keeping the state as a table entry can also let the FSM do some more advanced work, such as managing level of detail or culling. If your renderables have a state of potentially visible, then you can run the culling checks on just these entities rather than those not even up for consideration this frame. Using count lodding with FSMs allows for game flow logic as allowing the triggering of a game state change to emit the state's linked entities could provide a good point for debugging any problems. Also, using condition tables makes it very easy to change the reactions and add new alphabet without having to change a lot of code.

6.2 Implementing Transitions

Finite state machines normally have an input stream that modifies the internal state as fast as new signals arrive. This can lead to very fast state switching, but to maintain the parallelism, transform oriented finite state machines only run one update based on all signals available at the time of the tick. There is no ambiguity however, as if there are more than one conditional responses matching the input state, then it is fine to go to more than one state at once as a result.

Consider the finite state machine in a point and click adventure, where the character has to find three objects. In most games, the logic would be defined as a state waiting for all objects to be found. In a condition table finite state machine, there could be three different tables representing each of the objects, and while any table is populated, the game will not progress.

In a transform oriented finite state machine, the transitions are generated by the condition tables transforming the input signals into table insertions and deletions. Whenever a condition is matched, it emits an entity ID and a bool into a transition table (or a null if you intend to reduce the table in a separate process) ready for processing in the commit stage. This means that a finite state machine can react to multiple signals in a deterministic way because the state will not change before it has finished processing all the possible condition matches. There is no inherent temporal coupling in this design. Traditional finite state machines don't allow for multiple reactions at once as they transition from one state to another, naturally reacting on the order of signals, which is perfect for a finite state machine built for a lexer or parser, but possibly not for a generic gameplay state machine.

6.3 Condition tables as Triggers

Sometimes a finite state machine transition has side effects. Some finite state machines allow you to add callbacks on transitions, so you can attach onEnter, onExit, and onTransition effects. Because the transform oriented approach has a natural rhythm of state transforms to transition request transforms to new state, it's simple to hook into any state transition request table and add a little processing before the state table row modifications are committed.

You can use hooks for logging, telemetry, or game logic that is watching certain states. If you have a finite state machine for mapping input to player movement, it's important to have it react to a player state that adjusts the control method. For example, if

the player has different controls when under water, the onEnter of the inWater state table could change the player input mapping to allow for floating up or sinking down. It would also be a good idea to attach any oxygen-level gauge hooks here to. If there is meant to be a renderable for the oxygen-level, it should be created by the player entering the inWater state. It should also be hooked into the inWater onExit transition table, as it will probably want to reset or begin regenerating at that point.

One of the greatest benefits of being able to hook into transitions, is being able to keep track of changes in game state for syncing. If you are trying to keep a network game sycnronised, knowing exactly what happened and why can be the difference between a 5 minute and 5 day debugging session.

6.4 Conditions as Events

The entries in a condition table represent the events that are interesting, the ones that affect the flow of the game and the way in which the data is transformed. This means that the condition tables drive the flow, but they themselves are tables that can be manipulated, so they can self modify, which means we can use this technique to provide high perfomance data-driven, yet data-oriented development practices.

Condition tables are tables that can have rows inserted or deleted just like any other table. If the rows are added, modified, or removed based on the same rules as everything else, then you have a data-driven general purpose development model. One that doesn't even sacrifice performance for flexibility.

Chapter 7

Searching

When looking for specific data, it's very important to remember why you're doing it. If the search is not necessary, then that's your biggest possible saving. Finding if a row exists in a table will be slow if approached naively. You can manually add searching helpers such as binary trees, hash tables, or just keep your table sorted by using ordered insertion whenever you add to the table. If you're looking to do the latter, this could slow things down, as ordered inserts aren't normally concurrent, and adding extra helpers is normally a manual task. In this section we find ways to combat all these problems.

7.1 Indexes

Database management systems have long held the concept of indexes. They are traditionally automatically added when a database management system notices that a particular query has been run a large number of times. We can use this idea and implement a just-in-time indexing system in our games to provide the same kinds of performance improvement. Every time you want to find out if an element exists, or even just generate a subset, like when you need

to find all the entities in a certain range, you will have to build it as a pocess to generate a row or table, but the process that causes the row or table generation can be thought of as an object that can transform itself dependent on the situation. Starting out as a simple linear search query (if the data is not already sorted), the process can find out that it's being used quite often through internal telemetry, and proably be able to discover that it's generally returns a simply tunable set of results, such as the first N items in a sorted list, and can hook itself into the tables that it references. Hooking into the insertion, modification, and deletion would allow the query to update itself rather than run the full query again. This can be a significant saving, but it can also be useful as it is optional. As it's optional, it can be garbage collected.

If we build generalised back ends to handle building queries into these tables, they can provide multiple benefits. Not only can we provide garbage collection of indexes that aren't in use, but they can also make the programs in some way self documenting and self profiling. If we study the logs of what tables had pushed for building indexes for their queries, then we can quickly see data hotspots and where there is room for improvement. The holy grail here is self optimising code.

7.2 data-oriented Lookup

The first step in any data-oriented approach to searching is to understand the difference between the key, and the data that is dependent on the key. Object-oriented solutions to searching often ask the object whether or not it satisfies some criteria. Because the object is asked a question, there is often a large amount of code called, memory indirectly accessed, and cache-lines filled but hardly used. Even in a non-Object-oriented approach, there is frequently still a lot of cache abuse. Following is a listing for a simple binary search for a key in a naive implementation of an animation container. This kind of data access pattern is common in animation, hash lookups, and

other spatial searches.

```
struct AnimKey {
  float time;
  float x,y,z;
};
struct Anim {
  int numKeys;
  AnimKey *keys;
  AnimKey *GetKeyAtTime( float t ) {
    int l = 0, h = numKeys-1;
    int m = (l+h) / 2;
    while( l < h ) {
      if( keys[m].time > t )
        h = m;
      else
        l = m;
      m = (l+h) / 2;
    }
    return keys[i];
  }
};
```

Listing 7.1: Binary search for a given time value

We can improve on this very quickly by moving to a structure of arrays. What follows is a quick rewrite that saves us a lot of memory access.

```
struct AnimKey {
  float x,y,z;
};
struct Anim {
  int numKeys;
  float *keyTime;
  AnimKey *keys;
  AnimKey *GetKeyAtTime( float t ) {
    int l = 0, h = numKeys-1;
    int m = (l+h) / 2;
    while( l < h ) {
      if( keyTime[m] > t )
        h = m;
      else
        l = m;
      m = (l+h) / 2;
    }
    return keys[i];
  }
};
```

Listing 7.2: Binary search for a given time value

Using the data-oriented SoA, we speed this up, but why is it faster? The first impression most people get is that we've moved from a nice solid 16byte structure to a horrible or padded 12byte structure for the animation keys. Sometimes it pays to think a bit further than what looks right as first glance. Primarily, we are now finding the key by hunting for a value in a list of values. Not looking for a whole struct by one of its members in an array of structs. This is faster because the cache will be filled with mostly pertinent data. There are ways to organise the data better still, but any more optimisation requires a complexity or space time trade off. A basic binary search will home in on the correct data quite quickly, but each of the first steps will cause a new cache-line to be read in. If you know how big your cache-line is, then you can check all the values that have been loaded for free while you wait for the next cache-line to load in. Once you have got near the destination, most of the memory you need is in memory and all you're doing from then on is making sure you have found the right key. In a good SoA engine, all this can be done behind the scenes with a well optimised search algorithm usable all over the game code. It is worth mentioning again, that every time you break out into larger data structures, you deny your proven code the chance to be reused.

If the reason for a search is simpler, such as checking for existence, then there are even faster alternatives. Bloom filters offer a constant time lookup that even though it produces some false positives, can be tweaked to generate a reasonable answer hit rate for very large sets. In particular, if you are checking for which table an row exists in, then bloom filters work very well, by providing data about which tables to look in, usually only returning the correct table, but sometimes more than one.

In relational databases, indexes are added to tables at runtime when there are multiple queries that could benefit in their presence. For our data-oriented approach, there will always be some way to

speed up a search but only by looking at the data. If the data is not already sorted, then an index is a simple way to find the specific item we need. If the data is already sorted, but needs even faster access, then a search tree optimised for the cache-line size would help.

A binary search is one of the best search algorithms for using the smallest number of instructions to find a key value. But if you want the fastest algorithm, you must look at what takes time, and a lot of the time, it's not instructions. Loading a whole cache-line or information and doing as much as you can with that would be a lot more helpful than using the smallest number of instructions. Consider these two different data layouts for a animation key finding system:

```
1  struct Anim {
2    float keys[num];
3    VALUE_TYPE values[num];
4  };
5
6  struct KeyClump { float startTimes[32]; } // 128 bytes
7  keyClump root;
8  keyClump subs[32];
9  // implicit first element that has the same value as root.
       startTimes[n],
10 // but last element is the same as root.startTimes[n+1]
11 VALUE_TYPE Values[32*32];
```

Listing 7.3: Anim key formats

A binary search is quick, but if you know you are going to be doing completely random access of your data, but your data is of a certain size, you can make even more optimisations. In the key clumps version, the search loads the root (which has the times of every 32^{nd} key,) then after finding which subroot to load loads that subroot. Once these two pieces of data are memory, there is enough information to generate a lookup index for the two values stored in *Values*, and the sub key interval interpolation value.

```
1  VALUE_TYPE GetAtT( t )
2    int rootIndex = binaryFind( t, root );
3    KeyClump &theSub = subs[rootIndex];
4    int subIndex = binaryFind( t, theSub );
5    int keyIndex = rootIndex * 32 + subIndex;
```

```
6     float *keyTimes = &theSub;
7     const float t1 = keyTimes[subIndex];
8     const float t2 = keyTimes[subIndex+1];
9     float blend = (t-t1) / ( t2-t1 );
10    return Interp( Values[keyIndex], Values[keyIndex+1], blend );
11  }
```

Listing 7.4: Finding in the clump

But most data isn't this simple to optimise. A lot of the time, we have to work with spatial data, but because we use objects, it's hard to strap on an efficient spatial reference container after the fact. It's virtually impossible to add one at runtime to an externally defined class of objects.

Adding spatial partitioning when your data is in a simple data format like rows allows us to generate spatial containers or lookup systems that will be easy to maintain and optimise for each particular situation. Because of the inherent reusability in data-oriented transforms, we can write some very highly optimised routines for the high level programmers.

7.3 Finding lowest or highest is a sorting problem

In some circumstances you don't even really need to search. If the reason for searching is to find something within a range, such as finding the closest food, or shelter, or cover, then the problem isn't really one of searching, but one of sorting. In the first few runs of a query, the search might literally do a real search to find the results, but if it's run often enough, there is no reason not to promote the query to a runtime updating sorted subset of some other tables' data. If you need the nearest three elements, then you keep a sorted list of the nearest three elements, and when an element has an update, insertion or deletion, you can update the sorted three with that information. For insertions or modifications that bring elements that are not in the set closer, you can check that the element is closer and

pop the lowest before adding the new element to the sorted best. If there is a deletion, or a modification that makes one in the sorted set a contender for elimination, a quick check of the rest of the elements to find a new best set might be necessary. If you keep a larger than necessary set of best values, however, then you might find that this never happens.

```
Array<int> bigArray;
Array<int> bestValue;
const int LIMIT = 3;

void AddValue( int newValue ) {
  bigArray.push( newValue );
  bestValue.sortedinsert( newValue );
  if( bestValue.size() > LIMIT )
    bestValue.erase(bestValue.begin());
}
void RemoveValue( int deletedValue ) {
  bigArray.remove( deletedValue );
  bestValue.remove( deletedValue );
}
int GetBestValue() {
  if( bestValue.size() ) {
    return bestValue.top();
  } else {
    int best = bigArray.findbest();
    bestvalue.push( best );
    return best;
  }
}
```

Listing 7.5: keeping more than you need

The trick is to find, at runtime, the best value to use that covers the solution requirement. The only way to do that is to check the data at runtime. For this, either keep logs, or run the tests with dynamic resizing based on feedback from the table's query optimiser.

7.4 Finding random is a hash/tree issue

For some tables, the values change very often. For a tree representation to be high performance, it's best not to have a high number of modifications as each one could trigger the need for a rebalance. Of

course, if you do all your modifications in one stage of processing, then rebalance, and then all your reads in another, then you're probably going to be okay still using a tree. If you have many different queries on some data, you can end up with multiple different indexes. How frequently the entries are changed should influence how you store your index data. Keeping a tree around for each query could become expensive, but would be cheaper than a hash table in most implementations. Hash tables become cheaper where there are many modifications interspersed with lookups, trees are cheaper where the data is mostly static, or at least hangs around for a while. When the data becomes constant, a perfect hash can trump a tree. Trees can be quickly made from data, faster than hash tables which need you to run a hash function per entry being inserted. Hash tables can be quickly updated and can take up only marginally more space than trees if implemented with care. Perfect hash tables use precalculated hash functions to generate an index, and don't require any space other than is used to store the constants and the array of pointers or offsets into the original table. If you have the time, then you might find a perfect hash that returns the actual indexes. I'm not sure we have that long though.

For example, if we need to find the position of someone given their gamertag, the players won't be sorted by gamertag normally, so we need a gamertag to player lookup. This data is mostly constant during game so would be better suited to a tree or perfect hash if we could generate one fast enough. Generally, people leave games and join them, so a tree is probably as static a container as is sensible to use. Given the number of players isn't likely to go over a ten thousand at any point, a trie based off the gamertag hash value should provide very quick lookup.

A hash can be more useful when the table is more chaotic, for example, messages have senders and recipients. If we need to do something with all the messages from or to one sender or recipient, then at least one of those is not going to be the normal sorted order of the table. Because the table is in constant flux, keeping a tree is virtually useless, also the hash table variant can handle multiple

entries with the same key a lot easier than a tree implementation as it can use the standard linked list of entries in a hash bucket to collect all the messages with a certain sender/recipient.

Chapter 8

Sorting

For some subsystems, sorting is a highly important function. Sorting the primitive render calls so that they render front to back for opaque objects can have a massive impact on GPU performance, so it's worth doing. Sorting the primitive render calls so they render back to front for alpha blended objects is usually a necessity. Sorting sound channels by their amplitude over their sample position is a good indicator of priority.

Whatever you need to sort for, make sure you need to sort first, as usually, sorting is a highly memory intense business.

8.1 Do you need to?

There are some algorithms that seem to require sorted data, but don't, and some that require sorted data but don't seem to. Be sure you know whether you need to before you make any false moves.

One common use of sorting in games is in the render pass where some engine programmers recommend having all your render calls sorted by a high bit count key generated from a combination of depth, mesh, material, shader, and other flags such as whether the

call is alpha blended. This then allows the renderer to adjust the sort at runtime to get the most out of the bandwidth available. In the case of the rendering list sort, you could run the whole list through a general sorting algorithm, but in reality, there's no reason to sort the alpha blended objects with the opaque objects, so in many cases you can take a first step of putting the list into two separate buckets, and save some n for your O. Also, choose your sorting algorithm wisely. With opaque objects, the most important part is usually sorting by textures then by depth, but that can change with how much your fill rate is being trashed by overwriting the same pixel multiple times. If your overdraw doesn't matter too much but your texture uploads do, then you probably want to radix sort your calls. With alpha blended calls, you just have to sort by depth, so choose an algorithm that handles that case best, usually a quick sort or a merge sort bring about very low but guaranteed accurate sorting.

8.2 Maintain by insertion sort or parallel merge sort

Depending on what you need the list sorted for, you could sort while modifying. If the sort is for some AI function that cares about priority, then you may as well insertion sort as the base heuristic commonly has completely orthogonal inputs. If the inputs are related, then a post insertion table wide sort might be in order, but there's little call for a full scale sort.

If you really do need a full sort, then use an algorithm that likes being parallel. Merge sort and quick sort are somewhat serial in that they end or start with a single thread doing all the work, but there are variants that work well with multiple processing threads, and for small data sets there are special sorting network techniques that can be faster than better algorithms just because they fit the hardware so well[1].

[1]Tony Albrecht proves this point in his article on sorting networks

8.3 Sorting for your platfom

Always remembering that in data-oriented development you must look to the data before deciding which way you're going to write the code. What does the data look like? For rendering, there is a large amount of data with different axes for sorting. If your renderer is sorting by mesh and material, to reduce vertex and texture uploads, then the data will show that there are a number of render calls that share texture data, and a number of render calls that share vertex data. Finding out which way to sort first could be figured out by calculating the time it takes to upload a texture, how long it takes to upload a mesh, how many extra uploads are required for each, then calculating the total scene time, but mostly, profiling is the only way to be sure. If you want to be able to profile, or allow for runtime changes in case your game has such varying asset profiles that there is no one solution to fit all, having a flexible sorting criteria is extremely useful, and sometimes necessary. Fortunately, it can be made just as quick as any inflexible sorting technique, bar a small set up cost.

Radix sort is the fastest serial sort. If you can do it, radix sort is very fast because it generates a list of starting points for data of different values. This allows the sorter to drop their contents into containers based on a translation table, a table that returns an offset for a given data value. If you build a list from a known small value space, then radix sort can operate very fast to give a coarse first pass. The reason radix sort is serial, is that it has to modify the table it is reading from in order to update the offsets for the next element that will be put in the same bucket.

It is possible to make this last stage of the process parallel by having each sorter ignore any values that it reads that are outside its working set, meaning that each worker reads through the entire set of values gathering for their bucket, but there is still a small chance of non-linear performance due to having to write to nearby

---

http://seven-degrees-of-freedom.blogspot.co.uk/2010/07/question-of-sorts.html

memory on different threads. During the time the worker collects the elements for its bucket, it could be generating the counts for the next radix in the sequence, only requiring a summing before use in the next pass of the data, mitigating the cost of iterating over the whole set with every worker.

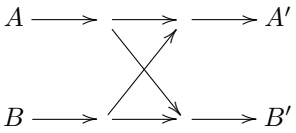
If your data is not simple enough to radix sort, you might be better off using a merge sort or a quick sort, but there are other sorts that work very well if you know the length of your sortable buffer at compile time, such as sorting networks. Through merge-sort is not itself a concurrent algorithm, the many early merges can be run in parallel, only the final merge is serial, and with a quick pre-parse of the to-be-merged data, you can finalise with two threads rather than one by starting from both ends (you need to make sure that the mergers don't run out of data). Though quick sort is not a concurrent algorithm each of the sub stages can be run in parallel. These algorithms are inherently serial, but can be turned into partially parallelisable algorithms with O(log n) latency.

When your n is small enough, a traditionally good technique is to write an in place bubble sort. The algorithm is so simple, it is hard to write wrong, and because of the small number of swaps required, the time taken to set up a better sort could be better spent elsewhere. Another argument for rewriting such trivial code is that inline implementations can be small enough for the whole of the data and the algorithm to fit in cache[2]. As the negative impact of the inefficiency of the bubble sort is negligible over such a small n, it is hardly ever frowned upon to do this.

If you've been developing data-oriented, you'll have a transform that takes a table of n and produces the sorted version of it. The algorithm doesn't have to be great to be better than bubble sort, but notice that it doesn't cost any time to use a better algorithm as the data is usually in the right shape already. Data-oriented development naturally leads us to reuse good algorithms.

[2]It might be wise to have some templated inline sort functions in your own utility header so you can utilise the benefits of miniaturistion, but don't drop in a bloated std::sort

Sorting networks work by implementing the sort in a static manner. They have input data and run swap if necessary functions on pairs of values of that input data before outputting the final. The simplest sorting network is two inputs.

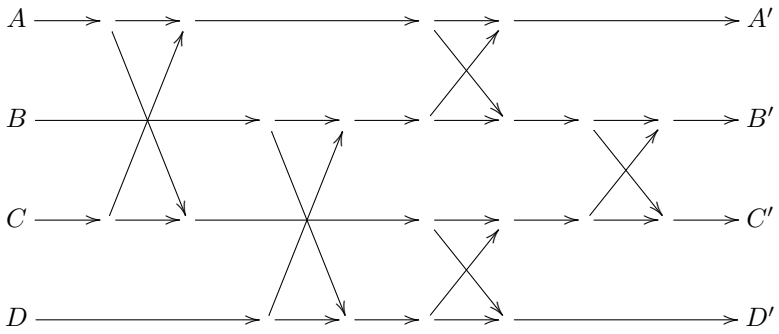


If the values entering are in order, the sorting crossover does nothing. If the values are out of order, then the sorting cross over causes the values to swap. This can be implemented as branchless writes:

```
a' <= MAX(a,b)
b' <= MIN(a,b)
```

This is fast on any hardware. The MAX and MIN functions will need different implementations for each platform and data type, but in general, branch free code executes faster than code that includes branches.

Introducing more elements:



What you notice here is that the critical path is not long (just three stages in total), and as these are all branch free functions, the performance is regular over all data permutations. With such a regular performance profile, we can use the sort in ways where the variability of sorting time length gets in the way, such as just-in-time sorting for sub sections of rendering. If we had radix sorted our renderables, we can network sort any final required ordering as we can guarantee a consistent timing.

Sorting networks are somewhat like predication, the branch free way of handling conditional calculations. Because sorting networks use a min / max function, rather than a conditional swap, they gain the same benefits when it comes to the actual sorting of individual elements. Given that sorting networks can be faster than radix sort for certain implementations, it goes without saying that for some types of calculation, predication, even long chains of it, will be faster than code that branches to save processing time. Just such an example exists in the Pitfalls of Object Oriented Design presentation, concluding that lazy evaluation costs more than the job it tried to avoid. I have no hard evidence for it yet, but I believe a lot of AI code could benefit the same, in that it would be wise to gather information even when you are not sure you need it, as gathering it might be quicker than deciding not to. For example, seeing if someone is in your field of vision, and is close enough, might be small enough that it can be done for all AI rather than just the ones that require it, or those that require it occasionally.

Chapter 9

Relational Databases

We now examine how it is possible to put game data into a database: First we must give ourselves a level file to work with. We're not going to go into the details of the lowest level of how we utilise large data primitives such as meshes, textures, sounds and such. For now, think of assets as being the same kind of data as strings[1].

We're going to define a level file for a game where you hunt for keys to unlock doors in order to get to the exit room. The level file will be a list of different game objects that exist in the game, and the relationships between the objects. First, we'll assume that it contains a list of rooms (some trapped, some not), with doors leading to other rooms which can be locked. It will also contain a set of pickups, some that let the player unlock doors, some that affect the player's stats (like health potions), and all the rooms have lovely textured meshes, as do all the pickups. One of the rooms is marked as the exit, and one has a player start point.

[1]There are existing APIs that present strings in various ways dependent on how they are used, for example the difference between human readable strings (usually UTF-8) and ascii strings for debugging. Adding sounds, textures, and meshes to this seems quite natural once you realise that all these things are resources for human consumption

```
// create rooms, pickups, and other things.
m1 = mesh( "filename" )
m2 = mesh( "filename2" )
t1 = texture( "textureFile" )
t2 = texture( "textureFile2" )
k1 = pickup( TYPE_KEY, m5,t5, TintColourGold )
k2 = pickup( TYPE_POTION, m4,t4, null )
r1 = room( worldPos(0,0), m1, t1, hasPickup(k1)+hasTrap(true,10hp)
     +hasDoorTo(r2) )
r2 = room( worldPos(20,5), m2,t2, hasDoorTo(r3,lockedWith(k1)+
     isStart(worldPos(22,7) )
r3 = room( worldPos(30,-5), m3,t3, isExit() )
```

Listing 9.1: A setup script

In the first step, you take all the data from the object creation script, and fit it into rows in your table. To start with, we create a table for each type of object, and a row for each instance of those obejcts. We need to invent columns as necessary, and use NULL everywhere that an instance doesn't have that attribute/column.

Once we have taken the construction script and generated tables, we find that these tables contain a lot of nulls. The nulls in the rows replace the optional content of the objects. If an object instance doesn't have a certain attributes then we replace those features with nulls. When we plug this into a database, it will handle it just fine, but storing all the nulls seems unnecessary and looks like it's wasting space. Present day database technology has moved towards keeping nulls to the point that a lot of large, sparse table databases have many more null entries than they have data. They operate on these sparsely filled tables with functional programming techniques. We, however should first see how it worked in the past with relational databases. Back when SQL was first invented there were three known stages of data normalisation (there currently are six recognised numbered normal forms, plus BoyceCodd (which is a stricter variant of third normal form), and Domain Key (which, for the most part defines a normalisation which requires using domain knowledge instead of storing data).

Meshes

MeshID	MeshName
m1	"filename"
m2	"filename2"

Textures

TextureID	TextureName
t1	"texturefile"
t2	"texturefile2"

Pickups

MeshID	TextureID	PickupType	ColourTint	PickupID
m5	t5	KEY	Gold	k1
m6	t6	POTION	(null)	k2

Rooms

RoomID	MeshID	TextureID	WorldPos	Pickup1	Pickup2	...
r1	m1	t1	0,0	k1	k2	...
r2	m2	t2	20,5	(null)	(null)	...
r3	m3	t3	30,-5	(null)	(null)	...

...	Trapped	DoorTo	Locked	Start	Exit
...	10hp	r2	(null)	(null)	false
...	(null)	r3	k1	22,7	false
...	(null)	(null)	(null)	(null)	true

Table 9.1: Initial tables created by converting constructors

9.1 Normalisation

First normal form requires that every row be distinct and rely on the key. We haven't got any keys yet, so first we must find out what these are.

In any table, there is a set of columns that when compared to any other row, are unique. Databases guarantee that this is possible by not allowing complete duplicate rows to exist, there is always some differentiation. Because of this rule, every table has a key because every table can use all the columns at once as its primary key. For example, in the mesh table, the combination of meshID and filename is guaranteed to be unique. The same can be said for the textureID and filename in the textures table. In fact, if you use all the columns of the room table, you will find that it can be used to uniquely define that room (obviously really, as if it had the same key, it would in

fact be literaly describing the same room, but twice).

Going back to the mesh table, because we can guarantee that each meshID is unique in our system, we can reduce the key down to just one or the other or meshID or filename. We'll choose the meshID as it seems sensible, but we could have chosen the filename[2].

If we choose textureID, pickupID, and roomID as the primary keys for the other tables, we can then look at continuing on to first normal form.

Meshes

MeshID	MeshName
m1	"filename"
m2	"filename2"

Textures

TextureID	TextureName
t1	"texturefile"
t2	"texturefile2"

Pickups

PickupID	MeshID	TextureID	PickupType	ColourTint
k1	m5	t5	KEY	Gold
k2	m6	t6	POTION	(null)

Rooms

RoomID	MeshID	TextureID	WorldPos	Pickup1	Pickup2	...
r1	m1	t1	0,0	k1	k2	...
r2	m2	t2	20,5	(null)	(null)	...
r3	m3	t3	30,-5	(null)	(null)	...

...	Trapped	DoorTo	Locked	Start	Exit
...	10hp	r2	(null)	(null)	false
...	(null)	r3	k1	22,7	false
...	(null)	(null)	(null)	(null)	true

Table 9.2: primary keys in **bold**

9.1.1 1st Normal Form

First normal form can be described as making sure the tables are not sparse. We require that there be no null pointers and that there be no arrays of data in each element of data. This can be performed

[2]in fact, the filename is the primary key of the filedata on the storage media, filesystems are simple key-value databases with very large complex/large values

as a process of moving the repeats and all the optional content to other tables. Anywhere there is a null, it implies optional content. Our first fix is going to be the Pickups table, it has an optional ColourTint element. We invent a new table PickupTint, and use the primary key of the Pickup as the primary key of the new table.

Pickups

PickupID	MeshID	TextureID	PickupType
k1	m5	t5	KEY
k2	m6	t6	POTION

PickupTint

PickupID	**ColourTint**
k1	Gold

Table 9.3: splitting out the pickup tint

two things become evident at this point, firstly that normalisation appears to create more tables and less columns in each table, secondly that there are only rows for things that matter. The latter is very interesting as when using Object-oriented approaches, we assume that objects can have attributes, so check that they are not NULL before continuing, however, if we store data like this, then we know everything is not NULL. Let's move onto the Rooms table: make a new table for the optional Pickups, Doors, Traps, and Start.

We're not done yet, the RoomDoors table has a null in it, so we add a new table that looks the same as the existing RoomDoorTable, but remove the row with a null. We can now remove the Locked column from the RoomDoor table.

All done. But, first normal form also mentions no repeating groups of columns, meaning that the PickupTable is invalid as it has a pickup1 and pickup2 column. To get rid of these, we just need to move the data around like this:

Notice that now we don't have a unique primary key. This is an error in databases as they just have to have something to index with, something to identify a row against any other row. We are allowed to combine keys, and for this case we used the all column trivial key. Now, let's look at the final set of tables in 1NF:

Rooms

RoomID	MeshID	TextureID	WorldPos	Exit
r1	m1	t1	0,0	false
r2	m2	t2	20,5	false
r3	m3	t3	30,-5	true

Room Pickup

RoomID	Pickup1	Pickup2
r1	k1	k2

Rooms Doors

RoomID	DoorTo	Locked
r1	r2	(null)
r2	r3	k1

Rooms Traps

RoomID	Trapped
r1	10hp

Start Point

RoomID	Start
r2	22,7

Table 9.4: adding more tables to remove nulls

Rooms Doors

RoomID	DoorTo
r1	r2
r2	r3

Rooms Doors Locks

RoomID	DoorTo	Locked
r2	r3	k1

Table 9.5: locked doors as separate table

The data, in this format takes much less space in larger projects as the number of NULL entries would have only increased with increased complexity of the level file. Also, by laying out the data this way, we can add new features without having to revisit the original objects. For example, if we wanted to add monsters, normally we would not only have to add a new object for the monsters, but also add them to the room objects. In this format, all we need to do is add a new table:

And now we have information about the monster and what room it starts in without touching any of the original level data.

Room Pickup

RoomID	Pickup
r1	k1
r1	k2

Table 9.6: add more rows to reduce repeating columns

9.1.2 Reflections

What we see here as we normalise our data is a tendency to add information when it becomes necessary, but at a cost of the keys that associate the data with the entity. Looking at many third party engines and APIs, you can see parallels with the results of these normalisations. It's unlikely that the people involved in the design and evolution of these engines took their data and applied database normalisation techniques, but sometimes the separations between object and componets of objects can be obvious enough that you don't need a formal technique in order to realise some positive structural changes.

In some games the entity object is not just an object that can be anything, but is instead a specific subset of the types of entity involved in the game. For example, in one game there might be an object type for the player character, and one for each major type of enemy character, and another for vehicles. This object oriented approach puts a line, invisilbe to the player, but intrusive to the developer, between classes of object and instances. It is intrusive because every time a class definition is used instead of a differing data, the amount of code required to utilise that specific entity type in a given circumstance increases. In many codebases there is the concept of the player, and the player has different attributes to other entities, such as lacking AI controls, or having player controls, or having regenerating health, or having ammo. All these different traits can be inferred from data decisions, but sometimes they are made literal through code in classes. When these differences are put in code, interfacing between different classes becomes a game of adapting, a known design pattern that should be seen as a symptom

Rooms

RoomID	MeshID	TextureID	WorldPos	Exit
r1	m1	t1	0,0	false
r2	m2	t2	20,5	false
r3	m3	t3	30,-5	true

Room Pickup

RoomID	**Pickup**
r1	k1
r1	k2

Rooms Doors

RoomID	DoorTo
r1	r2
r2	r3

Rooms Doors Locks

RoomID	DoorTo	Locked
r2	r3	k1

Rooms Traps

RoomID	Trapped
r1	10hp

Start Point

RoomID	Start
r2	22,7

Table 9.7: final 1st normal form

Monster

MonsterID	Attack	HitPoints	StartRoom
M1	2	5	r1
M2	2	5	r3

Table 9.8: monster table

of mixed levels of specialisation in a set of classes. When developing a game, this ususally manifests as time spent writing out templated code that can operate on multiple classes rather than refactoring the classes involved into more discrete components. This could be considered wasted time as the likelyhood of other operations needing to operate on all the objects is greater than zero, and the effort to refactor into components is usually no greater than the effort to create a working templated operation.

9.1.3 Domain Key / Knowledge

Whereas first normal form is almost a mechanical process, 2nd Normal form and beyond become a little more investigative. It is required that the person doing the normalisation understand the data in order to determine whether parts of the data are dependent on the key and whether or not they would be better suited in a new table, or written out as a form of procedural result due to domain knowledge.

Domain key normal form is normally thought of as the last normal form, but for developing efficient data structures, it's one of the things that is best studied early and often. The term domain knowledge is preferrable when writing code as it makes more immediate sense and encourages use outside of keys and tables. Domain knowledge is the idea that data depends on other data, given information about the domain in which it resides. Domain knowledge can be as simple as knowing the collowquialism for something, such as knowing that a certain number of degrees Celsius or Fahrenheit is hot, or whether some SI unit relates to a man-made concept such as 100m/s being rather quick. These procedural solutions are present in some operating systems or applications: the progress dialogue that may say "about a minute" rather than an innaccurate and erratic seconds countdown. Hoever, domain knowledge isn't just about human interpretation of data. For example things such as boiling point, the speed of sound, of light, speed limits and average speed of traffic on a given road network, psychoacoustic properties, the boiling point of water, and how long it takes a human to react to any given visual input. All these facts may be useful in some way, but can only be put into an application if the programmer adds it specifically as procedural domain knowledge or as an attribute of a specific instance.

Domain knowledge is useful because it allows us to lose some otherwise unnecessarily stored data. It is a compiler's job to analyse the produced output of code (the abstract syntax tree) to then provide itself with data upon which it can infer and use domain knowledge about what operations can be omitted, reordered, or transformed

to produce faster or cheaper assembly. Profile guided optimisation is another way of using domain knowledge, the domain being the runtime, the knowledge being the statistics of instructino coverage and order of calling. PGO is used to tune the location of instructions and hint branch predictors so that they produce even better performance based on a real world use case.

9.1.4 All Normal Forms

First normal form: remove repeating columns and nulls by adding new tables for optional values or one to may relationships. That is, ensure that any potentially null columns in your table are moved to their own table and use the primary table's key as their primary key. Move any repeating columns (such as item1, item2, item3) to a separate single table (item) and again use the original table's primary key as the primary key for this new table.

Second, Third, and BC (Boyce-Codd) normal form: split tables such that changes to values only affect the minimum amount of tables. Make sure that columns rely on only the table key, and also all of the table key. Look at all the columns and be sure that each one relies on all of the key and not on any other column.

Fourth normal form: ensure that your tables columns are truly dependent on each other. An example might be keeping a table of potential colours and sizes of clothing, with an entry for each colour and size combination. Most clothes come in a set of sizes, and a set of colours, and don't restrict certain colours based on size so there would be no omissions from a matrix of all the known sizes and all the known colours, instead the table should be two tables, both with the garment as primary key, with size or colour as the data. This holds for specifications, but doesn't hold for the case where an item needs to be checked for whether or not it is in stock. Not all sizes and colour combinations may be in stock, so an "in-stock" cache table would be right to have all three columns.

Fifth normal form: if the values in multiple columns are related by possibility, then remove the columns out into the separate ta-

bles of possible combinations. For example, an Ork can use simple weapons or orcish weapons, a human can use simple weapons or human weapons, if we say that Human Arthur is using a stick, then the stick doesn't need the column specifying that it is an simple weapon, but equally, if we say the Ork KruelGut is using a sword, we don't need a column specifying that he is using an orcish sword, as he cannot use a human sword and that is the only other form of sword available. In this case then we have a table that tells us what each entity is using for a weapon (we know what race each entity is in another table), and we have a table for what weapon types are available per race, and the weapon table can still have both weapontype and actual weapon. The benefit of this is that the entities only need to maintain a lower range "weapon" and not also the weapontype as the weapontype can be inferred from the weapontypes available for the race. This reduction in memory useage may come at a cost of performance, or may increase performance depending on how the data is accessed.

Sixth normal form: This is an uncommon normal form because it is one that is normally only important for keeping track of changing states and can cause an explosion in the number of tables. However, it can be useful in reducing memory usage when tracking rapidly changing independent columns of an otherwise fully normalised table. For example, if a character is otherwise fully normalised, but they need to keep track of how much money, XP, and spell points they have over time, it might be better to split out the money, XP, and spell points columns into character/time/value tables that can be separately updated without causing a whole new character record to be written purely for the sake of time stamping.

Domain/Key normal form: using domain knowledge remove columns or parts of column data because it depends on some other colum through some intrinsic quality of the domain in which the table resides. Information provided by the data may already be available in other data. For eaxmple the colloquialism of old-man is dependent on having data on the gender column and age colum, and can be inferred. Thus it doesn't need to be in a column and instead can

live as a process. Equally, foreshadowing existence based processing here, a dead ork has zero health and a unhurt ork has zero damage. If we maintain a table for partially damaged orks health values then we don't need storage space undamaged ork health. This minor saving can add up to large savings when operating on multiple entities with multiple stats that can vary from a known default.

9.1.5 Sum up

At this point we can see it is perfectly reasonable to store any highly complex data structures in a database format, even game data with its high interconnectedness and rapid design changing criteria. What is still of concern is the less logical and more binary forms of data such as material data, mesh data, audio clips and runtime integration with scripting systems and control flow.

Platform specific resources such as audio, texture, vertex or video formats are opaque to most developers, and moving towards a table oriented approach isn't going to change that. In databases, it's common to find column types of boolean, number, and string, and when building a database that can handle all the natural atomic elements of our engine, it's reasonable to assume that we can add new column types for these platform or engine intrinsics. The textureID and meshID column used in room examples could be smart pointers to resources. Creating new meshes and textures may be as simple as inserting a new row with the asset URL and keeping track of whether or not an entry has arrived in the fulfilled streaming request table or the URL to assetID table that could be populated by a behind the scenes task scheduler.

As for integration with scripting systems and using tables for logic and control flow, chapters on finite state machines, existence based processing, condition tables and hierarchical level of detail show how tables don't complicate, but instead provide opportunity to do more with fewer resources as results flow without constraint normally associated with object or entity linked data structures.

9.2 Implicit Entities

Getting your data out of a database and into your objects can appear quite daunting, but a database approach to data storage does provide a way to allow old executables to run off new data, it also allows new executables to run off old data, which is can be vital when working with other people who might need to run an earlier or later version. We saw that sometimes adding new features required nothing more than adding a new table. That's a non-intrusive modification if you are using a database, but a significant change if you're adding a new member to a class. If you're trying to keep your internal code object-oriented, then loading up tables to generate your objects isn't as easy as having your instances generated from script calls, or loading them in a binary file and doing some pointer fixup. Saving can be even more of a nightmare as when going from objects back to rows, it's hard to tell if the database needs rows added, deleted, or just modified. This is the same set of issues we normally come across when using source control. The problem with figuring out what was moved, deleted, or added when doing a three way merge.

But again, this is only true if you convert your database formatted files into explicit objects. These denormalised objects are troublesome because they are in some sense, self aware. They are aware of what their type is, and what to a large degree, what their type could be. A problem seen many times in development is where a base type needs to change because a new requirement does not fit with the original vision. Explicit objects act like the original tables before normalisation: they have a lot of nulls when features are not used. If you have a reasonable amount of object instances, this wasted space can cause problems.

You can see how an originally small object can grow to a disproportionate size, how a Car class can gain 8 wheel pointers in case it is a truck, how it can gain up to 15 passengers, optionally having lights, doors, sunroof, collision mesh, different engine, a driver base class pointer, 2 trailers, and maybe a physics model base class

pointer so you can switch to different physics models when level collision is or isn't present. All these are small additions, not considered bad in themselves, but every time you check a pointer for null it can turn out either way. If you guess wrong, or the branch predictor guesses wrong, the best you can hope for is an instruction cache miss or pipeline flush before doing an action. Pipeline flushes may not be very expensive on out-of-order CPUs, but the PS3 and Xbox360 have in-order CPUs and suffer a stall when this happens.

It is generally assumed that if you bind your type into an explicit object, you also allow for runtime polymorphism. This is seen as a good thing, in that you can keep the over arching game code smaller and easier to understand. You can have a main loop that runs over all of your objects, calling: "Think", "Update", "Render", and then you loop forever. But runtime polymorphism is not only made possible by using explicit types in an Object-oriented system. In the set of tables we just normalised, the Pickups were optionally coloured. In traditional Object-oriented C++ games development we generally define the Pickups, and then either have an optional component for tinting or derive from the Pickup type and add more information to the GetColour member function. In either case it can be necessary to override the base class in some fashion. You can either add the optional tint explicitly or make a new class and change the Pickup factory to accept that some Pickups can have colour tinting. In our database normalisation, adding tinting required only adding a new table and populating it with only the Pickups that were tinted.

The more commonly seen as a bad approach (littering with pointers to optional things) is problematic in that it leads to a lot of cases where you cannot quickly find out whether an object needs special care unless you check this extended type info. For example, when rendering the Pickups, for each one in every frame, we might need to find out whether to render it in the alpha blend pass or in the solid pass. This can be an important question, and because it is asked every frame in most cases, it can add some dark matter to our profile, some cold code ineffciency.

The more commonly seen as correct approach (that is create a new type and adjust the factory), has a limitation that you cannot change a Pickup from non-tinted to tinted at runtime without some major changes to how you use Pickups in the code. This means that if an object is tinted, it remains tinted unless you have the ability to clone and swap in the new object. This seems quite strict in comparison to the pointer littering technique. You may find that you end up making all pickups tinted in the end, just because they might be tinted at some time in the future. This would mean that you would have the old code for handling the untinted pickup rotting while you assume it still works. This has been the kind of code that causes one in a thousand errors that are very hard to debug unless you get lucky.

The database approach maintains the runtime dynamicity of the pointer littering approach by allowing for the creation of tint rows at runtime post creation. It also maintains the non-littering aspect of the derived type approach because we didn't add anything to any base type to add new functionality. It's quicker than both for iterating, which in our example was binning into alpha blend render pass and solid render pass. By only iterating over the table of tints picking out which have alpha values that cause it to be binned in alpha blended, we save memory accesses as we're only traversing the list of potentially alpha blended rather than running over all the objects. All objects not in the TintedPickup set but in the Pickup set can be assumed to be solid rendered, no checking of anything required. We have one more benefit with the separation of data in that it is easier to prefetch rows for analysis when they are much simpler in layout, and we could likely have more rows in the cache at once than in either of the other methods.

The fundamental difference between the database style approach and the object-oriented approach is how the type of a Pickup is defined. Traditionally, we derive new explicit types to allow new functionality or behaviours. With databases, we add new tables and into them insert rows referencing existing entities to show new information about old information. We imply new functionality while not

explicitly adding anything to the original form.

In an explicit entity system, the noun is central, the entity has to be extended with data and functions. In an implicit entity system, the adjectives are central, they refer to an entity, implying its existence by recognising it as being their operand. The entity only exists when there is something to say about it. By contrast, in an explicit entity system, you can only say something about an entity that already exists and caters for that describability.

9.3 Components imply entities

If we go back to our level file, we see that the room table is quite explicit about itself. What we have is a simple list of rooms with all the attributes that are connected to the room written out along the columns. Although adding new features is simple enough, modification or reusing any of these separately would prove tricky.

If we change the room table into multiple adjective tables, we can see how it is possible to build it out of components without having a literal core room. We will add tables for renderable, position, and exit

Room Renderable

RoomID	MeshID	TextureID
r1	m1	t1
r2	m2	t2
r3	m3	t3

Room Position

RoomID	WorldPos
r1	0,0
r2	20,5
r3	30,-5

Exit Room

RoomID
r3

Table 9.9: new room tables

With the new layout there is no core room. It is only implied

through the fact that it is being rendered, and that it has a position. In fact, the last table has taken advantage of the fact that the room is now implicit, and changed from a bool representing whether the room is an exit room or not into a one column table. The entry in that table implies the room is the exit room. This will give us an immediate memory usage balance of one bool per room compared to one row per exit room. Also, it becomes slightly easier to calculate if the player is in an exit room, they simply check for that room's existence in the ExitRoom table and nothing more.

Another benefit of implicit entities is the non-intrusive linking of existing entities. In our data file, there is no space for multiple doors per room. With the pointer littering technique, having multiple doors would require an array of doors, or maybe a linked list of doors, but with the database style table of doors, we can add more rows whenever a room needs a new door.

Doors

RoomID	bfRoomID
r1	r2
r2	r3

Locked Doors

RoomID	bfRoomID	Locked
r2	r3	k1

Table 9.10: Doors between rooms

Notice that we have to make the primary key the combination of the two rooms. If we had kept the single room ID as a primary key, then we could only have one row per room. In this case we have allowed for only one door per combination of rooms. We can guarantee in our game that a room will only have one door that leads to another room; no double doors into other rooms. Because of that, the locks also have to switch to a compound key to reference locked doors. All this is good because it means the system is extending, but by changing very little. If we decided that we did need multiple doors from one room to another we could extend it thus:

There is one sticking point here, first normal form dictates that

Doors

DoorID	RoomID	RoomID
d1	r1	r2
d2	r2	r3

Locked Doors

DoorID	Locked
d2	k1

Table 9.11: Multiple doors between rooms

we should build the table of Doors differently, where we have two columns of doors, it requires that we have one, but multiple rows per door.

Doors

DoorID	RoomID
d1	r1
d1	r2
d2	r2
d2	r3

Locked Doors

DoorID	Locked
d2	k1

Table 9.12: 1st normal form doors

we can choose to use this or ignore it, but only because we know that doors have two and only two rooms. A good reason to ignore it could be that we assume the door opens into the first door. Thus these two room IDs actually need to be given slightly different names, such as *DoorInto*, and *DoorFrom*.

Sometimes, like with RoomPickups, the rows in a table can be thought of as instances of objects. In RoomPickups, the rows represent an instance of a PickupType in a particular Room. This is a many to many relationship, and it can be useful in various places, even when the two types are the same, such as in the RoomDoors table.

When most programmers begin building an entity system they

start by creating an entity type and an entity manager. The entity type contains information on what components are connected to it, and this implies a core entity that exists beyond what the components say about it. Adding information about what components an entity has directly into the core entity might help debugging for a little while, while the components are all still centred about entities, but it becomes cumbersome when we realise the potential of splitting and merging entities and have to move away from using an explicit entity inspector. Entities as a replacement for classes, have their place, and they can simplify the move from class central thinking to a more data-oriented approach.

9.4 Cosmic Hierarchies

Whatever you call them, be it Cosmic Base Class, Root of all Evil, Gotcha #97, or CObject, having a base class that everything derives from has pretty much been a universal failure point in large C++ projects. The language does not naturally support introspection or duck typing, so it has difficultly utilising CObjects effectively. If we have a database driven approach, the idea of a cosmic base class might make a subtle entrance right at the beginning by appearing as the entity to which all other components are adjectives about, thus not letting anything be anything other than an entity. Although component–based engines can often be found sporting an EntityID as their owner, not all require owners. Not all have only one owner. When you normalise databases, you find that you have a collection of different entity types. In our level file example we saw how the objects we started with turned into a MeshID, TextureID, RoomID, and a PickupID. We even saw the emergence through necessity of a DoorID. If we pile all these Ids into a central EntityID, the system should work fine, but it's not a necessary step. A lot of entity systems do take this approach, but as is the case with most movements, the first swing away swings too far. The balance is to be found in practical examples of data normalisation provided by the database

industry.

9.5 Structs of Arrays

In addition to all the other benefits of keeping your runtime data in a database style format, there is the opportunity to take advantage of structures of arrays rather than arrays of structures. SoA has been coined as a term to describe an access pattern for object data. It is okay to keep hot and cold data side by side in an SoA object as data is pulled into the cache by necessity rather than by accidental physical location.

If your animation timekey/value class resembles this:

```
struct Keyframe
{
  float time, x,y,z;
};
struct Stream
{
  Keyframe *keyframes;
};
```

Listing 9.2: animation timekey/value class

then when you iterate over a large collection of them, all the data has to be pulled into the cache at once. If we assume that a cacheline is 128 bytes, and the size of floats is 4 bytes, the Keyframe struct is 16 bytes. This means that every time you look up a key time, you accidentally pull in four keys and all the associated keyframe data. If you are doing a binary search of a 128 key stream, that could mean you end up loading 128bytes of data and only using 4 bytes of it in up to 6 chops. If you change the data layout so that the searching takes place in one array, and the data is stored separately, then you get structures that look like this:

```
struct KeyData
{
  float x,y,z;
};
struct stream
```

```
6 {
7   float *times;
8   KeyData *values;
9 };
```

Listing 9.3: struct of arrays

Doing this means that for a 128 key stream, a binary search is going to pull in at most three out of four cachelines, and the data lookup is guaranteed to only require one.

Database technology also saw this, it's called column oriented databases and the provide better throughput for data processing over traditional row oriented relational databases simply because irrelevant data is not loaded when doing column aggregations or filtering.

9.6 Stream Processing

Now we realise that all the game data and game runtime can be implemented in a database oriented approach, there's one more interesting side effect: data as streams. Our persistent storage is a database, our runtime data is in the same format as it was on disk, what do we benefit from this? Databases can be thought of as collections of rows, or collections of columns, but there is one more way to think about the tables, they are sets. The set is the set of all possible permutations of the attributes. For example, in the case of RoomPickups, the table is defined as the set of all possible combinations of Rooms and Pickups. If a room has a Pickup, then the bit is set for that room,pickup combination. If you know that there are only N rooms and M types of Pickup, then the set can be defined as a bit field as that is N*M bits long. For most applications, using a bitfield to represent a table would be wasteful, as the set size quickly grows out of scope of any hardware, but it can be interesting to note what this means from a processing point of view. Processing a set, transforming it into another set, can be thought of as traversing the set and producing the output set, but the interesting attribute

of a set is that it is unordered. An unordered list can be trivially parallel processed. There are massive benefits to be had by taking advantage of this trivialisation of parallelism wherever possible, and we normally cannot get near this because of the data layout of the object-oriented approaches.

Coming at this from another angle, graphics cards vendors have been pushing in this direction for many years, and we now need to think in this way for game logic too. We can process lots of data quickly as long as we utilise about stream processing as much as possible and use random access processing as little as possible. Stream processing in this case means to process data without having variables that are external to the current datum., thus ensuring the processes or transforms are trivially parallelisable.

When you prepare a primitive render for a graphics card, you set up constants such as the transform matrix, the texture binding, any lighting values, or which shader you want to run. When you come to run the shader, each vertex and pixel may have its own scratch pad of local variables, but they never write to globals or refer to a global scratchpad. Enforcing this, we can guarantee parallelism because the order of operations are ensured to be irrelevant. If a shader was allowed to write to globals, there would be locking, or it would become an inherently serial operation. Neither of these are good for massive core count devices like graphics cards, so that has been a self imposed limit and an important factor in their design.

Doing all processing this way, without globals / global scratchpads, gives you the rigidity of intention to highly parallelise your processing and make it easier to think about the system, inspect it, debug it, and extend it or interrupt it to hook in new features. If you know the order doesn't matter, it's very easy to rerun any tests or transforms that have caused bad state.

9.7 Beautiful Homogeneity

Apart from all the speed increases and the simplicity of extension, there is also an implicit tendency to turn out accidentally reusable solutions to problems. This is caused by the data being formatted much more rigidly, and therefore when it fits, can almost be seen as a type of duck-typing. If the data can fit a transform, a transform can act on it. Some would argue that just because the types match, doesn't mean the function will create the expected outcome, but this is simply avoidable by not reusing code you don't understand. Because the code becomes much more penetrable, it takes less time to look at what a transform is doing before committing to reusing it in your own code.

Another benefit that comes from the data being built in the same way each time, handled with transforms and always being held in the same types of container is that there is a very good chance that there are multiple intention agnostic optimisations that can be applied to every part of the code. General purpose sorting, counting, searches and spatial awareness systems can be attached to new data without calling for OOP adapters or implementing interfaces so that Strategies can run over them.

A final reason to work with data in an immutable way comes in the form of preparations for optimisation. C++, as a language, provides a lot of ways for the programmer to shoot themselves in the foot, and one of the best is that pointers to memory can cause unexpected side effects when used without caution. Consider this piece of code:

```
1 char buffer[ 100 ];
2 buffer[0] = 'X';
3 memcpy( buffer+1, buffer, 98 );
4 buffer[ 99 ] = '\0';
```

Listing 9.4: byte copying

this is perfectly correct code if you just want to get a string with 99 'X's in it. However, because this is possible, memcpy has to copy one byte at a time. To speed up copying, you normally load in a

lot of memory locations at once, then save them out once they are all in the cache. If your input data can be modified by your output buffer, then you have to tread very carefully. Now consider this:

```
int q=10;
int p[10];
for( int i = 0; i < q; ++i )
  p[i] = i;
```

Listing 9.5: trivially parallelisable code

The compiler can figure out that q is unaffected, and can happily unroll this loop or replace the check against q with a register value. However, looking at this code instead:

```
void foo( int* p, const int &q )
{
  for( int i = 0; i < q; ++i)
    p[i] = i;
}

int q=10;
int p[10];
foo( p, q );
```

Listing 9.6: potentially aliased int

The compiler cannot tell that q is unaffected by operations on p, so it has to store p and reload q every time it checks the end of the loop. This is called aliasing, where the address of two variables that are in use are not known to be different, so to ensure correctly functioning code, the variables have to be handled as if they might be at the same address.

Chapter 10

Optimisations and Implementations

When optimising software, you have to know what is causing the software to run slower than you need it to run. We find in most cases, data movement is what really costs us the most. In the GPU, we find it labelled under fill rate, and when on CPU, we call it cache-misses. Data movement is where most of the energy goes when processing data, not in calculating solutions to functions, or from running an algorithm on the data, but actually the fulfillment of the request for the data in the first place. As this is most definitely true about our current architectures, we find that implicit or calculable information is much more useful than cached values or explicit state data.

If we start our game development by organising our data in normalised tables, we have many opportunities for optimisation. Starting with such a problem agnostic layout, we can pick and choose from tools we've created for other tasks, at worst elevating the solution to a template or a strategy, before applying it to both the old and new use cases.

10.1 Tables

To keep things simple, advice from multiple sources indicates that keeping your data as vectors has a lot of positive benefits. There are reasons to not use STL, including extended compile and link times, as well as issues with memory allocations. Whether you use std::vector, or roll your own dynamicly sized array, it is a good starting place for any future optimisations. Most of the processing you will do will be transforming one array into another, or modifying a table in place. In both these cases, a simple array will suffice for most tasks.

For the benefit of your cache, structs of arrays can be more cache friendly if the data is not related. It's important to remember that this is only true when the data is not meant to be accessed all at once, as one advocate of the data-oriented design movement assumed that structures of arrays were intrinsically cache friendly, then put the x,y, and z coordinates in separate arrays of floats. The reason that this is not cache friendly should be relatively easy to spot. If you need to access the x,y, or z of an element in an array, then you more than likely need to access the other two axes as well. This means that for every element he would have been loading three cache-lines of float data, not one. This is why it is important to think about where the data is coming from, how it is related, and how it will be used. Data-oriented design is not just a set of simple rules to convert from one style to another.

If you use dynamic arrays, and you need to delete elements from them, and these tables refer to each other through some IDs, then you may need a way to splice the tables together in order to process them. If the tables are sorted by the same value, then it can be written out as a simple merge operation, such as in Listing 10.1.

This works as long as the == operator knows about the table types and can find the specific column to check against, and as long as the tables are sorted based on this same column. But what about the case where the tables are zipped together without being the sorted by the same columns? For example, if you have a lot of

```
Table<Type1> t1Table;
Table<Type2> t2Table;
Table<Type3> t3Table;

ProcessJoin( Func functionToCall ) {
  TableIterator A = t1Table.begin();
  TableIterator B = t2Table.begin();
  TableIterator C = t3Table.begin();
  while( !A.finished && !B.finished && !C.finished ) {
    if( A == B && B == C ) {
      functionToCall( A, B, C );
      ++A; ++B; ++C;
    } else {
      if( A < B || A < C ) ++A;
      if( B < A || B < C ) ++B;
      if( C < A || C < B ) ++C;
    }
  }
}
```

Listing 10.1: Zipping together multiple tables by merging

entities that refer to a modelID, and you have a lot of mesh-texture combinations that refer to the same modelID, then you will likely need to zip together the matching rows for the orientation of the entity, the modelID in the entity render data, and the mesh and texture combinations in the models. The simplest way to program a solution to this is to loop through each table in turn looking for matches such as in Listing 10.2.

Another thing you have to learn about when working with data that is joined on different columns is the use of join strategies. In databases, a join strategy is used to reduce the total number of operations when querying across multiple tables. When joining tables on a column (or key made up of multiple columns), you have a number of choices about how you approach the problem. In our trivial coded attempt you can see we simply iterate over the whole table for each table involved in the join, which ends up being $O(nmo)$ or $O(n^3)$ for rougly same size tables. This is no good for large tables, but for small ones it's fine. You have to know your data to decide

```
Table<Orientation> oTable;
Table<EntityRenderable> erTable;
Table<MeshAndTexture> modelAssets;

ProcessJoin( Func functionToCall ) {
  TableIterator A = oTable.begin();
  while( !A.finished() ) {
    TableIterator B = erTable.begin();
    while( !B.finished() ) {
      TableIterator C = modelAssets.begin();
      while( !C.finished() ) {
        if( A == B && B == C ) {
          functionToCall( A, B, C );
        }
       ++C;
      }
    ++B;
    }
  ++A;
  }
}
```

Listing 10.2: Join by looping through all tables

whether your tables are big[1] or not. If your tables are too big to use such a trivial join, then you will need an alternative strategy.

You can join by iteration, or you can join by lookup[2], or you can even join once and keep a join cache around.

The first thing could do is use the ability to have tables sorted in multiple ways at the same time. Though this seems impossible, it's perfectly feasible to add auxilary data that will allow for traversal of a table in a different order. We do this the same way databases allow for any number of indexes into a table. Each index is created and kept up to date as the table is modified. In our case, we implement each index the way we need to. Maybe some tables are written to in bursts, and an insertion sort would be slow, it might be better to

[1] dependent on the target hardware, how many rows and columns, and whether you want the process to run without trashing too much cache

[2] often a lookup join is called a join by hash, but as we know our data, we can use better row search algorithms than a hash when they are available

sort on first read. In other cases, the sorting might be better done on write, as the writes are infrequent, or always interleaved with reads.

Concatenation trees provide a quick way to traverse a list. Conc-trees usually are only minimally slower than a linear array due to the nature of the structure. A conc-tree is a high level structure that points to a low level structure, and many elements can pass through a process before the next leaf needs to be loaded. The code for a conc-tree doesn't remain in memory all the time like other list handling code, as the list offloads to an array iteration whenever it is able. This alone means that sparse conc-trees end up spending little time in their own code, and offer the benefit of not having to rebuild when an element goes missing from the middle of the array.

In addition to using concatenation trees to provide a standard iterator for a constantly modified data store, it can also be used as a way of storing multiple views into data. For example, perhaps there is a set of tables that are all the same data, and they all need to be processed, but they are stored as different tables for some reason, such as what team they are on. Using the same conc-tree code, they can be iterated as a full collection with any code that accepts a conc-tree instead of an array iterator.

Modifying a class can be difficult, especially at runtime when you can't affect the offsets in running code without at least stopping the process and updating the executable. Adding new elements to a class can be very useful, and in languages that allow it, such as Ruby and Python, and even Javascript, it can be used as a substitute for virtual functions and compositing. In other languages we can add new functions, new data, and use them at runtime. In C++, we cannot change existing classes, unless they are defined by some other mechanism than the compile time structure syntax. We can add elements if we define a class by its schema, a data driven representation of the elements in the class. The benefit of this is that schema can include more, or less, than normal given a new context. For example, in the case of some merging where the merging happens on two different axes, there could be two different schema

representing the same class. The indexes would be different. One schema could include both indexes, with which it would build up an combination table that included the first two tables merged, by the first index, but maintaining the same ordering so that when merging the merged table with the third table to be merged, the second index can be used to maintain efficient combination.

```
A, B, C
B has index(AB), and index(BC)
merge A,B (by index(AB))
AB, C
merge AB, C (by index(BC))
ABC
```

10.2 Transforms

Taking the concept of schemas another step, a static schema definition can allow for a different approach to iterators. Instead of iterating over a container, giving access to an element, a schema iterator can become an accessor for a set of tables, meaning the merging work can be done during iteration, generating a context upon which the transform operates. This would benefit large, complex merges that do little with the data, as there would be less memory usage creating temporary tables. It would not benefit complex transforms as it would reduce the likelihood that the next set of data is in memory ready for the next cycle.

For large jobs, a smarter iterator will help in task stealing, the concept of taking work away from a process that is already running in order to finish the job faster. A scheduler, or job management system, built for such situations, would monitor how tasks were progressing and split remaining work amongst any idle processors. Sometimes this will happen because other processes took less time to finish than expected, sometimes because a single task just takes longer than expected. Whatever the reason, a transform based design makes task stealing much simpler than the standard sequential

model, and provides a mechanism by which many tasks can be made significantly more parallel.

Another aspect of transforms is the separation of what from how, the separation of the loading of data to transform from the code that performs the operations on the data. In some languages, introducing map and reduce is part of the basic syllabus, in c++, not so much. This is probably because lists aren't part of the base language, and without that, it's hard to introduce powerful tools that require an understanding of them. These tools, map and reduce, can be the basis of a purely transform and flow driven program. Turning a large set of data into a single result sounds eminently serial, however, as long as one of the steps, the reduce step, is either associative, or commutative, then you can reduce in parallel for a significant portion of the reduction.

A simple reduce, one made to create a final total from a mapping that produces values of zero or one for all matching elements, can be processed as a less and less parallel tree of reductions. In the first step, all reductions produce the total of all odd-even pairs of elements, and produce a new list that goes through the same process. This list reduction continues until there is only one item left remaining. Of course this particular reduction is of very little use, as each reduction is so trivial, you'd be better off assigning an Nth of the workload to each of the N cores and doing one final summing. A more complex, but equally useful reduction would be the concatenation of a chain of matrices. Matrices are associative even if they are not commutative, and as such, the chain can be reduced in parallel the same way building the total worked. By maintaining the order during reduction you can apply parallel processing to many things that would normally seem serial as long as they are associative in the reduce step. Not only matrix concatenation, but also products of floating point values such as colour modulation by multiple causes such as light, diffuse, or gameplay related tinting. Building text strings can be associative, as can be building lists and of course conc-trees themselves.

10.3 Spatial sets for collisions

In collision detection, there is often a broad-phase step which can massively reduce the number of potential collisions we check against. When ray casting, it's often useful to find the potential intersection via an octree, bsp, or other single query accelerator. When running path finding, sometimes it's useful to look up local nodes to help choose a starting node for your journey.

All spatial data-stores accelerate queries by letting them do less. They are based on some spatial criteria and return a reduced set that is shorter and thus less expensive to transform into new data.

Existing libraries that support spatial partitioning have to try to work with arbitrary structures, but because all our data is already organised by table, writing adaptors for any possible table layout is simple. Writing generic algorithms becomes very easy without any of the side effects normally associated with writing code that is used in multiple places. Using the table based approach, because of its intention agnosticism (that is, the spatial system has no idea it's being used on data that doesn't technically belong in space), we can use spatial partitioning algorithms in unexpected places, such as assigning audio channels by not only their distance from the listener, but also their volume and importance. Making a 5 dimensional spatial partitioning system, or an N dimensional one, would only have to be written once, have unit tests written once, before it could be used and trusted to do some very strange things. Spatially partitioning by the quest progress for tasks to do seems a little overkill, but getting the set of all nearby interesting entities by their location, threat, and reward, seems like something an AI might consider useful.

10.4 Lazy Evaluation for the masses

When optimising Objectoriented code, it's quite common to find local caches of calculations done hidden in mutable member variables.

One trick found in most updating hierarchies is the dirty bit, the flag that says whether the child or parent members of a tree imply that this object needs updating. When traversing the hierarchy, this dirty bit causes branching based on data that has only just loaded, usually meaning there is no chance to guess the outcome and thus in most cases, causes a pipeline flush and an instruction lookup.

If your calculation is expensive, then you might not want to go the route that renderer engines now use. In render engines, it's cheaper to do every scene matrix concatenation every frame than it is only doing the ones necessary and figuring out if they are.

For example, in the /emphGCAP 2009 - Pitfalls of Object Oriented Programming presentation by Tony Albrecht in the early slides he declares that checking a dirty flag is less useful than not checking it as if it does fail (the case where the object is not dirty) the calculation that would have taken 12 cycles is dwarfed by the cost of a branch misprediction (23-24 cycles).

If your calculation is expensive, you don't want to bog down the game with a large number of checks to see if the value needs updating. This is the point at which existence-based-processing comes into its own again as existence the dirty table implies that it needs updating, and as a dirty element is updated it can be pushing new dirty elements onto the end of the table, even prefetching if it can improve bandwidth.

10.5 Not getting what you didn't ask for

When you normalise your data you reduce the chance of another multifaceted problem of object-oriented development. C++'s implementation of objects forces unrelated data to share cache-lines.

Objects collect their data by the class, but many objects, by design, contain more than one role's worth of data. This is partially because object-oriented development doesn't naturally allow for objects to be recomposed based on their role in a transaction, and partially because C++ needed to provide a method by which

you could have object-oriented programming while keeping the system level memory allocations overloadable in a simple way. Most classes contain more than just the bare minimum, partially because of inheritance, and partially because of the many contexts in which an object can play a part. Unless you have very carefully laid out a class, many operations that require only a small amount of information from the class, will load a lot of unnecessary data into the cache in order to do so. Only using a very small amount of the loaded data is one of the most common sins of the object-oriented programmer.

Every virtual call loads in the cache-line that contains the virtual-table pointer of the instance. If the function doesn't use any of the class's early data, then that will be cacheline usage in the region of only 4%. That's a memory throughput waste, and cannot be recovered without rethinking how you dispatch your functions. After the function has loaded, the program has to load the data it wants to work on, which can be scattered across the memory allocated for the class too. Sometimes you might organise the data so that the function is accessing contiguous blocks of useful information, but someone going in and adding new data at the top of the class can shift all the data that was previously on one cache-line, into a position where it is split over two, or worse, make the whole class unaligned causing virtually random timing properties on every call. This could be even worse if what they have added is a precondition that loads data from an unrelated area of memory, in which case it would require that the load finished before it even got to the pipeline flush if it failed the branch into loading the function body and the definitely necessary transformable data.

A general approach with the table formatted data is to preparse the table to produce a job list. There would be one job list per transform type identified by the data if it was to emulate a virtual call. Then, once built, transform using these new job tables to drive the process. For each transform type , run the transform over all the jobs in the job queue built for this transform. This is much faster as the function used to transform the data is no longer coming from a virtual lookup, but instead is implied by which table is being

processed, meaning no instruction cache misses after the first call to the transform function, and not loading the transform if there are zero entries.

Another benefit is that the required data can be reorganised now that it's obvious what data is necessary. This leads to simple to optimise data structures compared to an opaque class that to some extent, pretends that the underlying memory configuration is unimportant and encapsulated. This can help with pre-fetching and write combining to give near optimal performance without any low level coding.

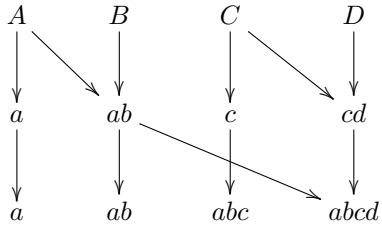
10.6 Varying Length Sets

Throughout the techniques so far, there's been an implied table structure to the data. Each row being a struct, or each table being a row of columns of data, depending on the need of the transforms. When we normally do stream processing, for example, with shaders, we normally use fixed size buffers. Most work done with stream processing has this same limitation, we tend to have a fixed number of elements for both sides. However, we saw that conc-trees solve the problem of mapping from one size input to another size output, and that it works by concatenating the output data into a cache-oblivious structure. This structure is very general purpose and could be a basis for preliminary work with map-reduce programming for your game, but there are cases where much better structures exist.

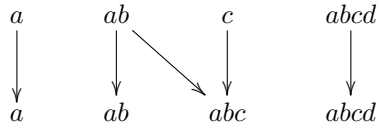
For filtering, that is where the input is known to be superset of the output, then there can be a strong case for an annealing structure. Like the conc-trees, each transform thread has its own output, but instead of concatenating, the reduce step would first generate a total and a start position for each reduce entry and then process the list of reduces onto the final contiguous memory.

If the filtering was a stage in a radix sort or something that uses a similar histogram for generating offsets, then a parallel prefix sum would reduce the time to generate the offsets. A prefix sum is the

running total of a list of values. The radix sort output histogram is a great example because the bucket counts indicate the starting points through the sum of all histogram buckets that come prior. $o_n = \sum_{i=0}^{n-1} b_i$. This is easy to generate in serial form, but in parallel we have to consider the minimum required operations to produce the final result. In this case we can remember that the longest chain will be the value of the last offset, which is a sum of all the elements. This is normally optimised by summing in a binary tree fashion. Dividing and conquering: first summing all odd numbered slots with all even numbered slots, then doing the same, but for only the outputs of the previous stage.



Then once you have the last element, backfill all the other elements you didn't finish on your way to making the last element. When you come to write this in code, you will find that these back filled values can be done in parallel while making the longest chain. They have no dependency on the final value so can be given over to another process, or managed by some clever use of SIMD.



Also, for cases where the entity count can rise and fall, you need a way of adding and deleting without causing any hiccups. For this, if you intend to transform your data in place, you need to handle the case where one thread can be reading and using the data that

you're deleting. To do this in a system where objects' existence was based on their memory being allocated, it would be very hard to delete objects that were being referenced by other transforms. You could use smart pointers, but in a multi-threaded environment, smart pointers cost a mutex to be thread safe for every reference and dereference. This is a high cost to pay, so how do we avoid it?

Don't ever delete.

Deletion is for wimps. If you are deleting in a system that is constantly changing, then you would normally use pools anyway. By explicitly not deleting, but doing something else instead, you change the way all code accesses data. You change what the data represents. If you need an entity to exist, such as a CarDriverAI, then it can stack up on your table of CarDriverAIs while it's in use, but the moment it's not in use, it won't get deleted, but instead marked as not used. This is not the same as deleting, because you're saying that the entity is still valid, won't crash your transform, but can be skipped as if it were not there until you get around to overwriting it with the latest request for a CarDriverAI. Keeping dead entities around is as cheap as keeping pools for your components.

10.7 Bit twiddling decision tables

Condition tables normally operate as arrays of condition flag bit fields. The flags are the collected results of conditions on the data. But, if the bit fields are organised by decision rather than by row, then you can call in only the necessary conditions into different decision transforms.

If the bits are organised by condition, then you can run a short transform on the whole collection of condition bits to create a new, simpler list of whether or nots. For example, you may have conditions for decisions that you can map to an alphabet of a-m, but only need some for making a decision. Imagine you need to be sure that a,d,f,g are all false, but e,j,k need to all be true. Given this logic, you can build a new condition, let's say q, that equals

$(a \wedge d \wedge f \wedge g) \wedge \neg(e \vee j \vee k)$ which can then be used immediately as a job list.

Organising the bits this way could be easier to parallelize as a transform that produces a condition stream would be in contention with other processes if they shared memory[3]. The benefit to row based conditions comes when the conditions change infrequently, and the number of things looking at the conditions to make decisions is small enough that they all fit in a single platform specific type, such as a 32bit or 64bit unsigned int. In that case, there would be no benefit in reducing the original contention when generating the condition bits, and because the number of views, or contexts about the conditions is low, then there is little to no benefit from splitting the processing so much.

If you have an entity that needs to reload when their ammo drops to zero, and they need to consider reloading their weapon if there is a lull in the action, then, even though both those conditions are based on ammo, the decision transforms are not based on the same condition. If the condition table has been generated as arrays of condition bitfields with each bit representing one row's state with respect to that condition, then you can halve the bandwidth to the check for definite-reload transform, and halve the bandwidth for the reload-consideration check. There will be one stream of bits for the condition of ammo equal to zero, and another stream for ammo less than max. There's no reason to read both, so we don't.

What we have here is another case of deciding whether to go with structures of arrays, or sticking with an array of structures. If we access the data from a few different contexts, then a structure of arrays trumps it. If we only have one context for using the conditions, then the array of structures, or in this case, array of masks, wins out. But remember to profile as any advice is only advice and only measuring can really provide proof, or evidence that assumptions are wrong.

[3]This would be from false sharing, or even real sharing in the case of different bits in the same byte

10.8 Joins as intersections

Sometimes, normalisation can mean you need to join tables together to create the right situation for a query. Unlike RDBMS queries, we can organise our queries much more carefully and use the algorithm from merge sort to help us zip together two tables. As an alternative, we don't have to output to a table, it could be a pass through transform that takes more than one table and generates a new stream into another transform. For example, per entityRenderable, join with entityPosition by entityID, to transform with AddRenderCall(Renderable, Position)

10.9 Data driven techniques

Apart from finite state machines there are some other common forms of data driven coding practices, some of which are not very obvious, such as callbacks, and some of which are very much so, such as scripting. In both these cases, data causing the flow of code to change will cause the same kind of cache and pipe-line problems as seen in virtual calls and finite state machines.

Callbacks can be made safer by using triggers from event subscription tables. Rather than have a callback that fires off when a job is done, have an event table for done jobs so that callbacks can be called once the whole run is finished. For example, if a scoring system has a callback from "badGuyDies", then in an Object-oriented message watcher you would have the scorer increment its internal score whenever it received the message that a badGuyDies. Instead run each of the callbacks in the callback table once the whole set of badguys has been checked for death. If you do that, and execute every time all the badGuys have had their tick, then you can add points once for all badGuys killed. That means one read for the internal state, and one write. Much better than multiple reads and writes accumulating a final score.

For scripting, if you have scripts that run over multiple entities,

consider how the graphics kernels operate with branches, sometimes using predication and doing both sides of a branch before selecting a solution. This would allow you to reduce the number of branches caused merely by interpreting the script on demand. If you go one step further an actually build SIMD into the scripting core, then you might find that you can run script for a very large number of entities compared to traditional per entity serial scripting. If your SIMD operations operate over the whole collection of entities, then you will pay almost no price for script interpretation[4].

[4]Take a look at the section headed *The Massively Vectorized Virtual Machine* on the bitsquid blog http://bitsquid.blogspot.co.uk/2012/10/a-data-oriented-data-driven-system-for.html

Chapter 11

Concurrency

Any book on games development practices for contemporary and future hardware must cover the issues of concurrency. There will come a time when we have more cores in our computers than we have pixels on screen, and when that happens, it would be best if we were already coding for it, coding in a style that allows for maximal throughput with the smallest latency. Thinking about how to solve problems for five, ten or even one hundred cores isn't going to keep you safe. You must think about how your algorithms would work when you have an infinite number of cores. Can you make your algorithms work for N cores?

Writing concurrent software has been seen as a hard task in the past because most people think they understand threading and can't get their heads around all the different corner cases that are introduced when you share the same memory as another thread. Fixing these with mutexs and critical sections can become a minefield of badly written code that works only 99% of the time. For any real concurrent development we have to start thinking about our code transforming data. Every time you get a deadlock or a race condition in threaded code, it's because there's some ownership issue. If you code from a data transform point of view, then there are some

simple ground rules that provide very stable tools for developing truly concurrent software.

11.1 What it means to be thread-safe

Academics consistently focus on what is possible and correct, rather than what is practical and usable, which is why we've been inundated with multi-threaded techniques that work, but cause a lot of unnecessary pain when used in a high performance system such as a game. The idea that something is thread-safe implies more than just that it is safe to use in a multi-threaded environment. There are lots of thread safe functions that aren't mentioned becuase they seem trivial, but it's a useful distinction to make when you are tracking down what could be causing a strange thread issue. There are functions without side-effects, such as the intrinsics for sin, sqrt, that return a value given a value. There is no way they can cause any other code to change behaviour, and no other code can change its behaviour either[1]. In addition to these very simple functions, there are the simple functions that change things in an idempotent fashion, such as memset.

Thread safe implies that it doesn't just access its own data, but accesses some shared data without causing the system to enter into an inconsistent state. Inconsistent state is a natural side effect of multiple processes accessing and writing to shared memory. It is these side effects that are the cause of many bugs in multi-threaded code, which is why the develpers of the Erlang language chose to limit the programmer to code that doesn't have side-effects. Any code that relies on reading from a shared memory before writing back an adjusted value can cause inconsistent state as there is no way to guarantee that the writing will take place before anyone else reads it before they modify it.

[1]If you discount the possibility of other code changing the floating point operation mode

```
int shared = 0;
void foo() {
int a = shared;
a += RunSomeCalculation();
shared = a;
}
```

Making this work in practice is hard and expensive. The standard technique used to ensure the state is consistent is to make the value update atomic. How this is achieved depends on the hardware and the compiler. Most hardware has an atomic instruction that can be used to create thread-safety through mutual exclusions. On most hardware the atomic instruction is a compare and swap, or CAS. Building larger tools for thread-safety from this has been the mainstay of multi-threaded programmers and operating system developers for decades, but with the advent of multi-core consoles, programmers not well versed in the potential pitfalls of multi-threaded development are suffering because of the learning cliff involved in making all their code work perfectly over six or more hardware threads.

Using mutual exclusions, it's possible to rewrite the previous example:

```
int shared = 0;
Mutex sharedMutex;
void foo() {
sharedMutex.acquire();
int a = shared;
a += RunSomeCalculation();
shared = a;
sharedMutex.release();
}
```

And now it works. No matter what, this function will now always finish its task without some other process damaging its data. Every time one of the hardware threads encounters this code, it stops all processing until the mutex is acquired. Once it's acquired, no other

hardware thread can enter into these instructions until the current thread releases the mutex at the far end.

Every time a thread-safe function uses a mutex section, the whole machine stops to do just one thing. Every time you do stuff inside a mutex, you make the code thread-safe by making it serial. Every time you use a mutex, you make your code run bad on infinite core machines.

Thread-safe, therefore, is another way of saying: not concurrent, but won't break anything. Concurrency is when multiple threads are doing their thing without any mutex calls, semaphores, or other form of serialisation of task. Concurrent means at the same time. A lot of the problems that are solved by academics using thread-safety to develop their multi-threaded applications needn't be mutex bound. There are many ways to skin a cat, and many ways to avoid a mutex. Mutex are only necessary when more than one thread shares write privileges on a piece of memory. If you can redesign your algorithms so they only ever require one thread to be given write privilege, then you can work towards a fully concurrent system.

Ownership is key to developing most concurrent algorithms. Concurrency only happens when the code cannot be in a bad state, not because it checks before doing work, but because the design is such that no process can interfere with another in any way.

11.2 Inherently concurrent operations

When working with tables of data, many operations are inherently concurrent. Simple transforms that take one table and generate the next step, such as those of physics systems or AI state / finite state machines, are inherently concurrent. You could provide a core per row / element and there would be no issues at all. Setting up the local bone transforms from a skeletal animation data stream, ticking timers, producing condition values for later use in condition tables. All these are completely concurrent tasks. Anything that could be implemented as a pixel or vertex shader is inherently concurrent,

which is why parallel processing languages such as shader models, do not cheaply allow for random write to memory, and don't allow accumulators across elements.

Seeing that these operations are inherently concurrent, we can start to see that it's possible to restructure our game from an end result perspective. We can use the idea of a structured query to help us find our critical path back to the game state. Many table transforms can be split up into much smaller pieces, possibly thinking along the lines of map reduce, bringing some previously serial operations into the concurrent solution set.

Any N to N transform is perfectly concurrent. Any N to ¡=N is perfectly concurrent, but depending on how you handle output NULLs, you could end up wasting memory. A reduce stage is necessary, but that could be managed by a gathering task after the main task has finished, or at least, after the first results have started coming in.

Any uncoupled transforms can be run concurrently. Ticking all the finite state machines can happen at the same time as updating the phsyics model, can happen at the same time as the graphics culling system building the next frame's render list. As long as all your different game state transforms are independent from each other's current output, they can be dependent on each other's original state and still maintain concurrency.

For example, the physics system can update while the renderer and the AI rely on the positions and velocities of the current frame. The AI can update while the animation system can rely on the previous set of states.

Multi stage transforms, such as physics engine broadphase, detection, reaction and resolution, will traditionally be run in series while the transforms inside each stage run concurrently. Standard solutions to these stages require that all data be finished processing from each previous stage, but if you can find some splitting planes for the elements during the first stage, you can then remove temporal cohesion from the processing because you can know what is necessary for the next step and hand out jobs from finished subsets

of each stage's results.

Concurrent operation assumes that each core operating on the data is free to access that data without interfering, but there is a way that seemingly unconnected processes can end up getting in each other's way. Most systems have multiple layers of cache, and this is where the accident can happen. False sharing is when data, though actually unrelated, is connected by the physical layout of the hardware. For example, the cachelines of a CPU might be 16 to 128 bytes long. If two different CPUs try to write to neighbouring bytes, or words, at best the cores will lock up as the memory is shunted around trying to keep memory consistent, but worse, could end up losing data if the cache is not coherent[2]. To reduce the chance of this happening, processing of small element tables should consider this and split any cooperation into cacheline size jobs.

11.3 Queues as gateways, and "Now"

When you don't know how many items you are going to get out of a transform, such as when you filter a table to find only the X that are Y, you need run a reduce on the output to make the table non-sparse. Doing this can be log(N) latent, that is pairing up rows takes serial time based on log(N). But, if you are generating more data than your input data, then you need to handle it very differently. Mapping a table out onto a larger set can be managed by generating into gateways, which once the whole map operation is over, can provide input to the reduce stage. The gateways are queues that are only ever written to by the Mapping functions, and only ever read by the gathering tasks such as Reduce. there is always one gateway per Map runtime. That way, there can be no sharing across threads. A Map can write that it has put up more data, and can set the content of that data, but it cannot delete it or mark any written data as having been read. The gateway manages a read head,

[2]cache coherency is the system by which changed data is propagated to other caches so that consistent state is achieved.

that can be compared with the write head to find out if there is any waiting elements. Given this, a gathering gateway or reduce gateway can be made by cycling through all known gateways and popping any data from the gateway read heads. This is a fully concurrent technique and would be just as at home in variable CPU timing solutions as it is in standard programming practices as it implies a consistent state through ownership of their respective parts. A write will stall until the read has allowed space in the queue. A read will either return "no data" or stall until the write head shows that there is something more to read.

When it comes to combining all the data into a final output, sometimes it's not worth recombining it into a different shape, in which case we can use conc-trees, a technique that allows for cache-friendly transforming while maintaining most of the efficiency of contiguous arrays. Conc-trees are a tree representation of data that almost singlehandedly allows for cache oblivious efficient storage of arbitrary lists of data. Cache oblivious algorithms don't need to know the size of the cache in order to fully utilise them and normally consist of some kind of divide and conquer core algorithm, and conc-trees do this by either representing the null set, a concatenation, or some data. If we use conc trees with output from transforms, we can optimise the data that is being concatenated so it fits well in cache lines, and we can also do the same with conc tree concatenate nodes, making them concatenate more than just two nodes, thus saving space and memory access. We can tune these structures per platform or by data type.

Given that we have a friendly structure for storing data, and that these can be built as concatenations rather than some data soup, we have basis for a nicely concurrent yet associative transform system. If we don't need to keep the associative nature of the transform, then we can optimise further, but as it comes at little to no cost, and most reduce operations rely on associativity, then it's good that we have that option.

Moving away from transforms, there is the issue of *now* that crops up whenever we talk about concurrent hardware. Sometimes,

when you are continually updating data in real time, not per frame, but actual real time, there is no safe time to get the data, or process it. Truly concurrent data analysis has to handle reading data that has literally only just arrived, or is in the process of being retired. data-oriented development helps us find a solution to this by not having a concept of now but a concept only of the data. If you are writing a network game, and you have a lot of messages coming in about a player, and their killers and victims, then to get accurate information about their state, you have to wait until they are already dead. With a data-oriented approach, you only try to generate the information that is needed, when it's needed. This saves trying to analyse packets to build some predicted state when the player is definitely not interesting, and gives more up to date information as it doesn't have to wait until the object representing the data has been updated before any of the most recent data can be seen or acted on.

The idea of now is present only in systems that are serial. There are multiple program counters when you have multiple cores, and when you have thousands of cores, there are thousands of different nows. Humans think of things happening at a certain time, but sometimes you can take so long doing something that more data has arrived by the time you're finished that you should probably go back and try again.

Take for example the idea of a game that tries to get the lowest possible latency between player control pad and avatar reaction. In a lot of games, you have to put up with the code reading the pad state, the pad state being used to adjust animations, the animations adjusting the renderables, the rendering system rastering and the raster system swapping buffers. In most games it takes at least three frames

read the player pad.
respond in logic to new pad state.
animate the character.
queueForRender the new bone positions.

FRAME happens, moves the render requests into the procesing list while calling:
renderToBackBuffer on the old lists.
FRAME happens again which starts to render our reaction to the pad state.
swapBuffers then shows our change as
VISIBLE

and many games have triple buffering and animation systems that don't update instantly.

In this case, the player could potentially be left until the last minute before checking pad status, and apply an emergency patchup to the in flight renderQueue. If you allowed a pad read to adjust the animation system's history rather than it's current state, you could request that it update it's historical commits to the render system, and potentially affect the next frame rather than the frame three buffer swaps from now. Alternatively, have some of the game run all potential player initiated events in parallel, then choose from the outcomes based on what really happened, then you can have the same response time but with less patching of data.

Chapter 12

In Practice

Data-oriented development is not rooted in theory, but practice. Because of this, it's hard to describe the methodology without some practical examples. In this chapter I will document some experiences with the data-oriented approach.

12.1 Data-manipulation

12.1.1 The Cube

In 22cans first experiment, there was call to handle a large amount of traffic from a large number of clients, potentially completely fragmented, and yet also completely syncronised. The plan was to develop a service capable of handling a large number of incoming packets of *update* data while also compiling it into compressed *output* data that could be send to the client all with a minimal turnaround time.

The system was developed in C++11 on Linux, and followed the basic tenets of data-oriented design from day one. The data was loaded, but not given true context. Manipulations on the data were procedures that operated on the data given other data. This was

a massive time saver when the servers needed a complete redesign. The speed of data processing was sufficient to allow us to run all the services in debug[1].

When the server went live, it wasn't the services that died, it was the front end. Nginx is amazing, but under that amount of load on a single server, with so many of the requests requiring a lock on an SQL db backend, the machine reached it's limit very quickly. For once, we think PHP itself wasn't to blame. We had to redesign all the services so they could work in three different situations so as to allow the server to become a distributed service. By not locking down data into contexts, it was relatively easy to change the way the data was processed, to reconfigure the single service that previously did all the data consumption, collation, and serving, into three different services that handled incoming data, merging the multiple instances, and serving the data on the instances. In part this was due to a very procedural approach, but it was also down to not linking together data that was separate by binding into an object context. The lack of binding allows for simpler recombination of procedures, simpler rewriting of procedures, and simpler repurposing of procedures from related services. This is something that object oriented approach makes harder because you can be easily tempted to start adding base classes and inheriting to gain common functionality or algorithms. As soon as you do that you start tying things together by what they mean rather than what they are, and then you lose the ability to reuse code.

12.1.2 Rendering order

While working on the in-house engine at Broadsword, we came up with an idea for how to reimplement the renderer that should have

[1]anyone remembering developing on a PS2 will likely attest to the minimal benefit you get from optimisations when your main bottleneck is the VU and GS. The same was true here, we had a simple bottleneck of the size of the data and the operations to run on it. Optimisations had minimal impact on processing speed, but they did impact our ability to debug the service when it did crash.

been much more efficient, not just saving CPU cycles, but also allowing for platform specific optimisations at run time by having the renderer analyse the current set of renderables and organise the whole list of jobs by what caused the least program changes, texture changes, constant changes and primitive render calls. The system seemed too good to be true, and though we tried to write the system, Broadsword never finished it. After Broadsword dissolved, when I did finish it, I didn't have access to a console[2], so I was only able to test out the performance on a PC. As expected the new system was faster, but only marginally. With hindsight, I now see that the engine was only marginally more efficient because the tests were being run on a system that would only see marginal improvements, but even the x86 architecture saw improvements, which can be explained away as slightly better control flow and generally better cache utilisation.

First let me explain the old system.

All renderables came from a scene graph. There could be multiple scene graph renders per frame, which is how the 2D and 3D layers of the game were composited, and each of them could use any viewport, but most of them just used the fullscreen viewport as the engine hadn't been used for a split screen game, and thus the code was probably not working for viewports anyway. Each scene graph render to viewport would walk the scene[3] collecting transforms, materials, colour tints, and meshes, to render, at which point the node that contained the mesh would push a render element onto the queue for rendering into that viewport. This queueing up of elements to render seemed simple and elegant at the time. It meant that programmers could quickly build up a scene and render it, most of the time using helpers that loaded up the assets and generated the right nodes for setting textures, transforms, shader constants, and meshes. These rendering queues were then sorted before rendering.

[2]The target platforms of the engine included the Playstation 2 and the Nintendo Wii along with Win32.

[3]sometimes multiple cameras belonged to the same scene to the scene graph would be walked multiple times

Sorted by material only for solid textures, and sorted back to front for alpha blended materials. This was the old days of fixed function pipelines and only minimal shader support on our target platforms. Once the rendering was done, all the calculated combinations were thrown away. This meant that for everything that was rendered, there was definitely a complete walk of the scene graph.

The new system, which was born before we were aware of data-oriented design, but was definitely born of looking at the data, was different in that it no longer required walking the scene graph. We wanted to maintain the same programmer friendly front edge API, so maintained the facade of a scene graph walk, but instead of walking the graph, we only added a new element to the rendering when the node was added, and only removed it when it was removed from the scene graph. This meant we had a lot of special code that looked for multiple elements registered in multiple view port lists, but, other than that, a render merely looked up into the particular node it cared about, gathered the latest data, and processed it pulling when it required it rather than being pushed things it didn't even care about.

The new system benefited from being a simple list of pointers from which to fetch data (the concept of a dirty transform was removed, all transforms were considered to be dirty every frame), and the computation was simplified for sorting as all the elements that were solid were already sorted from the previous render, and all the alpha blended elements were sorted because they belonged to the set of alpha blended elements. This lack of options accounted for some saved time, but the biggest saving probably came from the way the data was being gathered per frame rather than being generated from a incoherent tree. A tree that, to traverse, requried many pointer lookups into virtual tables as all the nodes were base classed to a base node type and all update and render calls were virtual causing many misses all the way through each of the tree walks.

In the end the engine was completely abandoned as Broadsword dissolved, but that was the first time we had taken an existing codebase and (though inadvertantly) converted it to data-oriented.

12.1.3 Keeping track of damage

During the development of the multiplayer code for Max Payne 3 there came a point where it became more and more important that we kept track of damage dealt to every player from every player. Sometimes this would be because we want to estimate more accurately how much health a player had. Sometimes this would be because we needed to know who saved who and thus who got a teammate saved bonus, or who got the assist. We kept a tight ship on Max Payne 3, we tried to keep the awards precise as far as we could go, no overcompensating for potential lost packets, no faking because we wanted a fair and competition grade experience where players could make all the difference through skill, and not through luck. To do this, we needed to have a system that kept track of all the bullets fired, but not fail when we missed some, or they arrived out of order. What we needed was a system that could be interrogated, but could receive data about things that had happened back in time. It needed to be able to heal itself in case of oddities, and last but not least, it needed to be really quick so we can interrogate it many times per frame.

The solution was a simple list of things that happened and at what time. Initially the data was organised as a list of structures with a sorted list of pointers to the data for quick queries about events over time, but after profiling the queries with and without the sorted list, the benefits of the lack of maintaining a sorted list outweighed the benefits of doing a one time sort on just the data needed to satisfy the query. There were only two queries, which were not frequently called, that benefitted from the sorted data, and when they were called they didn't need to be extremely fast as they were called as the result of player death, which not only happened less often than once a frame, but also regularly resulted in there being less to update as the player cared a lot less about visibility checks and handling network traffic[4]. This simple design allowed for

[4]dead men don't care about bullets and don't really care that much about what other players are doing

many different queries to run over the core data, and allowed for anyone to add another new query easily because there was no data-hiding getting in the way of any new kind of access. Object-oriented approaches to this kind of data handling often provide a gateway to the data but marshall it so that the queries become heavyweight and consistently targetted for optimisation. With a very simple data structure and open access to the data in any form, any new query could be as easily optimised as any existing one.

12.2 Game entities

12.2.1 Converting an object oriented player

While I was at Frontier Developments, I was working on a little space game that ended up being a boat game in caverns[5] and that was the first time I intentionally changed a game from object-oriented to component-oriented. The ships all had their rendering positions, their health, speed, momentum, ammo recharge timers etc, and these were all part of a base ship class that was extended to be a player class, and inherited from a basic element that was extended to also include the loot drops and the dangerous ice blocks. All this is pretty standard practice, and from the number of codebases I have seen, it's pretty light on the scale of inheritance with which games normally end up. Still, I had this new tool in my bag, the component-oriented approach specifically existence based processing. I wanted to try it out, see if it really did impact the profile of the game in any significant way. I started with the health values. I made a new component for health, and anyone that was at full health, or was dead, was not given an instance of the component. The player ship and the enemy ships all had no health values until they were shot and found wanting.

[5] due to concerns that a 2D space game might be misinterpretted as the next elite I imagine

Transitioning to the component based health took a while. Translating any game from one programming paradigm to another is not a task to be taken lightly, even when your game code is small and you've only spent sixty hours developing the game so far. Once the game was back up though, the profile spoke loud and clear of the benefits. I had previously spent some time in health code every frame, this was because the ships had health regenerators that ran and updated the health every time they got an update poll. They needed to do an update because they were updated, not because they needed to, and they didn't have any way to opt out before the componentisation, as the update to health was part of the ship update. Now the health update only ran over the active health instances, and removed itself once it reached max health rather than bounce of the max health.

This relatively small change allowed me to massively increase the number of small delicate ships in the game (ships that would beat the player ship by overrunning it rather than being able to withstand the player's fire and get close in to do some damage), and also lead to another optimisation: componentising the weapon recharge timer.

If the first change was a success, then the second was a major success. Instead of keeping track of when a weapon was ready to fire again, the time that it would be available was inserted into a sorted list of recharge times. Only the head of the list was checked every global update, meaning that a large number of weapons could be recharging at once and none of them caused any data access until they were very nearly or actually ready.

This immediate success lead me to believe the data-oriented design movement was really important and needed to be spread around, and probably caused my sudden hatred of object-oriented programming. From that point on, all I could see was cache-misses and pointless update checks.

12.2.2 People don't really exist.

A Mathematician, a Biologist and a Physicist are sitting in a street cafe watching people going in and coming out of the house on the other side of the street.

First they see two people going into the house. Time passes. After a while they notice three persons coming out of the house.

The Physicist: "The measurement wasn't accurate.".
The Biologists conclusion: "They have reproduced".
The Mathematician: "If now exactly 1 person enters the house then it will be empty again."

The followers in the GODUS prototype are little objects when they are running around, and though the code has changed quite a bit from the initial lists of pointers to people structures, and has thus become unweildy, there was one element that was a perfect example of hierarchical level of detail in the game logic. The followers, once they had started to build a building, dissappeared. They were no longer required in any way, so they were reduced to a mere increment of the number of people in a building. An object-oriented programmer disputed this being a good way to go, "but what if the person has a special weapon, or is a special character?". Logic prevailed. If there was a special weapon, then there would have been a count of those in the house, or the weapon would have become owned by the house, and as for a special character, the same kind of exception could be made, but in reality, which is where we firmly plant ourselves when developing in the data-oriented paradigm, these potential changes had not so far been requested, and by the end of the development, they still hadn't.

Another example from the GODUS prototype was the use of duck-typing. Instead of adding a base class for people and houses, just to see if they were meant to be under control of the local player, we used a templated function to extract a boolean value. Duck typing doesn't require anything more than certain member functions

or variables being available. They don't have to be in the same place, or come bundled into a base class: they just have to be present.

When teaching data-oriented design, I find the biggest hurdle is convincing programmers that data-oriented design is possible for some areas of development. It's very easy to accidentally assume that you need to bundle things into objects, especially after years of training and teaching object-oriented developlment. It won't come naturally and you will keep catching yourself building things as objects first then making them more relational afterwards. It will take time to fully remove the muscle memory of putting related things in the same class, but worry not, I'm sure all data-oriented developers go through this transition where they know they're not coding data-oriented, but they don't quite know how to do it yet.

12.2.3 Lazy evaluation molasses

The idea of lazy evaluation makes so much sense, and yet it's precisely that kind of sense that makes no sense for a computer. I remember there was a dirty bit check before an update in some of the global update code on Outsider, something I would have done a hundred times over before I started to get a better feel for what the computer was doing, and found one of the best lines-of-code to time-saved ratio fixes I ever found in a triple A game. The dirty bit check was just before the code that actually did the work of updating the instance. This meant that the CPU had to preload the *fixup* code while it was checking the dirty bit, and because the fixup code was virtual functions based, it meant that the code was loading the virtual table value and the function was preloaded all the while the dirty bit was about to say it wasn't necessary to do any of it. To make matters worse, the code was inside an if, implying to most compilers that it would be the more likely taken branch[6]. The fix was simple. Build up a local array of objects to actually

[6]profile guided optimisation might have saved a lot of time here, but the best solution is to give the profiler and optimiser a better chance by leaving them less to do

do an update of, then do them all at once. Because the code to do the update wasn't preloaded, the whole update took a lot less time. You can have your lazy evaluation, but don't preload the evaluation if you don't have to.

12.2.4 Component oriented design

While at Broadsword, I found an article about dungeon seige proclaiming the benefit of components when developing a game. The GOs or Game Objects that were used were components that could be stitched together to create new compound objects.

Taking that as inspiration, I looked at all our different scene-graph nodes, gameplay helper classes and functions we commonly used, and tried to distill them into their elements. I started by just making components such as *PlayerCharacter* and *AICharacter*, adding a *Prop* element before realising where I was going wrong. The object oriented mindset had told me to look at the compounds, not the elements, so I went back to the drawing board and started dissecting the objects again. Component oriented development works out best when you completely explode all your compounds, then recombine when you find the combinations are consistent, or when combining makes more sense to computation.

After I was done fully dissecting our basic game classes I had a long list of separate, elements that by themselves did very little[7]. At this point, I added the bootstrap code to generate the scene. Some components required other components at runtime, so I added a *requires* trait to the components, much like #include, except to make this work for teardown, I also added a count so components went away much like reference counted smart pointers folded away. The initial demo was a simple animated character running around a 3D environment, colliding with props via the collision Handler.

[7]Position, Velocity, PlayerInputMapper, VelocityFromInput, MeshRender, SkinnedMeshRender, MatrixPalette, AnimPlayer, AnimFromInput, and CollisionHandler

In the beginning, I had a container entity that maintained a list of components. Once I had a first working version, I stepped back and thought about the system. I started to see that the core entity wasn't really necessary. As long as any update components were updated, then the game would tick on. As long as the entity's components could find their requirement components, then the game would work. I shifted to having an implicit entity based on the UID I generate on entity creation. This meant that all entities were really only the components that were linked to that ID, as the ID didn't index into an array of entity objects, or point to an allocated entity, it was merely an ID off which to hang all the components.

Adding features to a class at runtime was now possible. I could inject an additional property into the existing entities because I had centred the entities around a key, and not any kind of central class. The next step from here was to add components via a scripting language so it would be possible to develop new gameplay code while running the game. Unfortunately this never happened, but some MMO engine developers have done precisely this.

The success of this demo convinced me that components would play a major role in our continuing game development, but alas, we only used the engine for one more product and the time to bring the new component based system up to speed for the size of the last project was estimated to be counter productive. Mick West released an article about how when he was at Neversoft, he did manage to convert the Tony Hawks code-base to components[8] which means it's not impossible to migrate, it's just not easy. If we were to start component oriented... well that's a different story, normally told by Scott Bilas.

[8] http://cowboyprogramming.com/2007/01/05/evolve-your-heirachy/

Chapter 13

Maintenance and reuse

When object-oriented design was first promoted, it advertised that it was easier to modify and extend existing code bases than the more traditional procedural approach. Though this is not true in practice, partially due to the language used by games developers and partially because there is an inherent coupling between objects with respect to their interfaces, it still remains a consideration of anyone who uses it when entering into another programming paradigm. Regardless of their level of expertise, an object-oriented programmer will cite the extensible, encapsulating nature of object-oriented development as a boon when it comes to working on larger projects.

Highly experienced but more objective developers have admitted or even written about how C++ is not highly suited to large scale development, but can be used in it as long as you follow strict guidelines. [LargeScaleC++] But for those that cannot immediately see the benefit of the data-oriented development paradigm with respect to maintenance, and evolutionary development, this chapter covers why it is easier than working with objects.

13.1 Debugging

The prime causes of bugs are the unexpected side effects of a transform, or an unexpected corner case where a conditional didn't return the correct value. In object-oriented programming, this can manifest in many ways, from an exception caused by de-referencing a null, to ignoring the interactions of the player because the game logic hadn't noticed it was meant to be interactive.

13.1.1 Lifetimes

One of the most common causes of the null dereference is when an object's lifetime is handled by a separate object to the one manipulating it. For example, if you are playing a game where the badguys can die, you have to be careful to update all the objects that are using them whenever the badguy gets deleted, otherwise you can end up dereferencing invalid memory which can lead to dereferencing null pointers because the class has destructed. data-oriented development tends towards this being impossible as the existence of an entity in a table implies its processability, and if you leave part of an entity around in a table, you haven't deleted the entity fully. This is a different kind of bug, but it's not a crash bug, and it's easier to find and kill as it's just making sure that when an entity is destroyed, all the tables it can be part of also destroy their elements too.

13.1.2 Avoiding pointers

When looking for data-oriented solutions to programming problems, we often find that pointers aren't required, and often make the solution harder to scale. Using pointers where null values are possible implies that each pointer doesn't only have the value of the object being pointed at, but also implies a boolean value for whether or not that instance exists. Removing this unnecessary extra feature can remove bugs, save time, and reduce complexity.

13.1.3 Bad State

Sometimes a bug is more to do with a game not being in the right state. Debugging then becomes a case of finding out how the game got into its current, broken state.

When you encapsulate your state, you hide internal changes. This quickly leads to adding lots of debugging logs. Instead of hiding, data-oriented suggests keeping data in simple forms, and potentially leaving it around longer than required can lead to highly simplified transform inspection. If you have a transform that appears to work, but for one odd case it doesn't, the simplicity of adding an assert and not deleting the input data reducing the amount of guesswork and toil required to generate the bug fix. If you keep most of your transforms as one-way, that is to say they take from one source, and produce or update another, but even if you run the code multiple times it will still produce the same results as it would the first time. The transform is idempotent. This useful property allows you to find a bug, then rewind and trace through without having to attempt to rebuild the initial state.

One way of keeping your code idempotent is to write your transforms in a single assignment style. If you operate with multiple transforms but all leading to predicated join points, you can guarantee yourself some timings, and you can look back at what caused the final state to turn out like it did without even rewinding. If your conditions are condition tables, just leave the inputs around until validity checks have been completed then you have the ability to go into any live system and check how it arrived at that state. This alone should reduce any investigation time to a minimum.

13.2 Reusability

A feature commonly cited by the object-oriented developers that seems to be missing from data-oriented development is reusability. The idea that you won't be able to take already written libraries

of code and use them again, or on multiple projects, because the design is partially within the implementation. To be sure, once you start optimising your code to the particular features of a software project, you do end up with unreusable code. While developing data-oriented projects, the assumed inability to reuse source code would be significant, but it is also highly unlikely. The truth is found when considering the true meaning of reusability.

Reusability is not fundamentally concerned in reusing source files or libraries. Reusability is the ability to maintain an investment in information. A wealth of knowledge for the entity that owns the development IP.

Copyright law has made it hard to see what resources have value in reuse, as it maintains the source as the object of it's discussion rather than the intellectual property represented by the source. The reason for this is that ideas cannot be copyrighted, so by maintaining this stance, the copyrighter keeps hold of this tenuous link to a right to withhold information. Reusability comes from being aware of the information contained within the medium it is stored. In our case, it is normally stored as source code, but the information is not the source code. With Object-Oriented development, the source can be adapted (adapter pattern) to any project we wish to venture. However, the source is not the information. The information is the order and existence of tasks that can and will be performed on the data. Viewing the information this way leads to an understanding that any reusability that a programming technique can provide comes down to it's mutability of inputs and outputs. It's willingness to adapt a set of temporally coupled tasks into a new usage framework is how you can find out how well it functions reusably.

In object-oriented development, you apply the information inherent in the code by adapting a class that does the job, or wrapper it, or use an agent. In data-oriented development, you copy the functions and schema and transform into and out of the input and output data structures around the time you apply the information contained in the data-oriented transform.

Even though, at first sight, data-oriented code doesn't appear as

resuable on the outside, the fact is that it maintains the same amount of information in a simpler form, so it's more reusable as it doesn't carry the baggage of related data or functions like object-oriented programming, and doesn't require complex transforms to generate the input and extract from the output like procedural progamming tends to generate due to the normalising.

Duck typing, not normally available in object-oriented programming due to a stricter set of rules on how to interface between data, can be implemented with templates to great effect, turning code that might not be obviously reusable into a simple strategy, or a sequence of transforms that can be applied to data or structures of any type, as long as they maintain a naming convention.

The object-oriented C++ idea of reusability is a mixture of information and architecture. Developing from a data-oriented transform centric viewpoint, architecture just seems like a lot of fluff code. The only good architecture that's worth saving is the actualisation of data-flow and transform. There are situations where an object-oriented module can be used again, but they are few and far between because of the inherent difficulty interfacing object-oriented projects with each other.

The most reusable object-oriented code appears as interfaces to agents into a much more complex system. The best example of an object-oriented approach that made everything easier to handle, was highly reusable, and was fully encapsulated was the `FILE` type from `stdio.h` that is used as an agent into whatever the platform and OS would need to open, access, write, and read to and from a file on the system.

13.3 Unit Testing

Unit testing can be very helpful when developing games, but because of the object-oriented paradigm making programmers think about code as representations of objects, and not as data transforms, it's hard to see what can be tested. Linking together unrelated concepts

into the same object and requiring complex setup state before a test can be carried out, has given unit testing a stunted start in games as object-oriented programming caused simple tests to be hard to write. Making tests is further complicated by the addition of the non-obvious nature of how objects are transformed when they represent entities in a game world. It can be very hard to write unit tests unless you've been working with them for a while, and the main point of unit tests is that someone that doesn't fully grok the system can make changes without falling foul of making things worse.

Unit testing is mostly useful during refactorings, taking a game or engine from one code and data layout into another one, ready for future changes. Usually this is done because the data is in the wrong shape, which in itself is harder to do if you normalise your data as you're more likely to have left the data in an unconfigured form. There will obviously be times when even normalised data is not sufficient, such as when the design of the game changes sufficient to render the original data-analysis incorrect, or at the very least, ineffective or inefficient.

Unit testing is simple with data-oriented technique because you are already concentrating on the transform. Generating tables of test data would be part of your development, so leaving some in as unit tests would be simple, if not part of the process of developing the game. Using unit tests to help guide the code could be considered to be partial following the test-driven development technique, a proven good way to generate efficient and clear code.

Remember, when you're doing data-oriented development your game is entirely driven by stateful data and stateless transforms. It is very simple to produce unit tests for your transforms. You don't even need a framework, just an input and output table and then a comparison function to check that the transform produced the right data.

13.4 Refactoring

During refactoring, it's always important to know that you've not broken anything by changing the code. Allowing for such simple unit testing gets you halfway there. Another advantage of data-orieted development is that, at every turn, it peels away the unnecessary elements, so you might find that refactoring is more a case of switching out the order of transforms more than changing how things are represented. Refactoring normally involves some new data representation, but as long as you normalise your data, there's going to be little need of that. When it is needed, tools for converting from one schema to another can be written once and used many times.

Chapter 14

Design Patterns

14.1 What caused design patterns

Design patterns came about as a way of writing down and transferring the knowledge gained from experience, originally in architectural design with the term coined as Patterns by Christopher Alexander. Software design patterns came about from experience developing object-oriented software. The new prefixed name was used in the title of the book *Design Patterns: elements of Reusable Object-Oriented Software* released in 1994, and has stuck ever since.

Using design patterns, it was found that the strategies and solutions to common activities and problems could be expressed as a pattern language of requirement and designs for structuring and the strategies for actions taken on them. This reusable design would be too complex to write as a templated function or a library of reusable code as it describes a solution that would normally require interfacing with and thinking about how it applies to your particular situation and code base. The reason why the solution is too complex to write is not because the problem being solved is complex in itself, but that the solution is normally rooted in the problem

domain rather than any specific programming language. However, the solution is only complex because it roots itself in the representation provided by the object-oriented language that was used to bring the design, in this case the problem domain itself, into the code in order for it to be solved. If the problem was not represented by objects but instead left as a technical design, then many design patterns would not be required. This is why when people talk about software design patterns they have to reference them as elements of reusable object-oriented design rather than elements of reusable software engineering.

14.2 Data-Oriented Design Patterns

Even though the original design patterns were elements of reusable Object-Oriented Design, there is scope for some pattern language for development given the Data-Oriented approach. Many of the sections so far have had an air of design pattern about them as they tend to solve generic problems with one or more generic solutions. This is where we gather and explain all the core patterns of development that are seen when working with a data-oriented code base, but always remember, a design pattern is a solution looking for a problem, and that alone is enough reason to be wary. In data-oriented design, we prefer to use design patterns as they were initially intended, as a way of communicating between engineers without needing to describe all the reasoning behind it. These architectural design patterns were a kind of written down common sense, which means in some sense, the data-oriented design patterns cannot be like these as common sense is often trumped by profiling or invention. Regardless, the ability to speak a common tongue is enough to merit trying to define some of the patterns present.

14.2.1 A to B transform

A stateless function that takes elements from one container and produces outputs into another container. All variables are constant over the lifetime of the transform.

The normal way of working and the best way to guarantee that your output and input tables don't clash, is to create a new table and transform into it. The A to B transform is also the basis of double buffered state and provides a powerful mechanism for enabling concurrent processing as you can ensure that A is read only, and B is write only for the whole time slice. The A to B transform is the transform performed by shaders and other compute kernels, and the same conditions apply: constants do not change, local variables only, globals are considered to be constants, data can be processed in any order and without any communication to any other element in the transform.

A to B transforms aren't expensive unless the row counts change, and even then you can mitigate those more expensive transforms with conc-trees and other in place transforms that reduce in the case of filtering transforms. There are two things to consider when transforming from one table to another:

Is this a transform to create new, or an updated state? Is this transform going to significantly change the values before they hit the output table?

The simplest transform is an update. This allows for concurrent state updates across multiple associated systems without coupling. The condition table transform is generally used to significantly change the values, putting them into a new table with a different schema, ready for decision transforms. Sometimes a transform will do more than one of these, such as when an update also emits rows into an event table ready for subscribing entities to react to a state change on the data.

An important requirement of the transform is that it is stateless. Transforms operate on data, use constants to adjust how they transform that data, and output in a standardised manner. Some amount

of restriction needs to be set up for the transforms to be worth doing, however, and one of those restraints is that the transform must be stateless. The data has state, but the transforms must not. If you look at how the design pattern works for existing streaming transform engines such as GPU shader models, you will see that the shader constants are uniform across all the elements in one render call, this allows for easy distribution of work across many different workers, even on different machines. You can have local working variables, but you cannot store out to some common location as that would introduce some coupling, potentially making one part of a kernel serial in an otherwise completely parallel process. Even accumulation over successive elements would mean that the input to one element depended on the output of another, which would immediately cause the whole process to become serial in nature. If you do this, you are going to break the fundamental parallelisable nature of the transform process, and with it, doom your code to running only as fast as one of your workers can handle the data.

14.2.2 In place transform

A stateless function that changes the data in a container based on constants alone.

There are times when that's not necessary to create a new table with a transform. In update calls for many tables, the update can in place modify rather than run as a double buffered state. The best way to in place modify can change depending on the layout of the data and what needs to be done with the data. Always consider how the data gets into the transform, and how it gets out.

For inputs, make sure you find the optimal way to organise the pre-fetching so that the transform is never starved of data, for outputs, make sure that you write as much as you can in tightly packed consecutive memory to offer opportunity for write combining. The majority of cases have the same restrictions and benefits as A to B transform.

If you are doing in place transforms, always remember to write

back once at most for each transform, otherwise aliasing rules can bite, for example, writing to a local accumulator when the type of the accumulator is the same as a type being read from, then aliasing may require your accumulator to be re-read every time it is updated. One way around this is to mark all your elements as restricted, thus ensuring that the compiler will assume it's safe to use the last value without reloading, but this might be vendor specific so test by looking at the assembly produced before declaring that the restrict will fix the problems inherent in accumulators or other load-modify-store values that may be used in in-place transforms.

Another thing to mention is that for both the in-place transform and the A to B transform, using a structure's static member is just as invalid as a global variable.

14.2.3 Generative transform

A function that generates data into an output container based on an algorithm rather than input data.

Sometimes all you need is some constants and only one small piece of data to generate a whole new table. If you've got to build a world from scratch, you can generate your map with procedural generators, but you have to seed a procedural generator and the small inputs or constants. These are what drives these types of transform. You don't need a large input set to drive a file loader, and that can be considered a generative transform. If you put filenames into a *load-me-please* table, and register a reaction transform into the *file-has-loaded* event table, then you have a generative function as your streaming engine. You can write your bootstrap as a generative transform that introduces elements into multiple tables. If this transform is a script, then you have the basis for a completely data-driven development. If instead of a filename you provide a seed and expect to get back a chunk of terrain, whether by pure procedural generation, or loading, or some combination, then you have the basis for a distributed processing landscape system. In the chapter on hierarchical level of detail, the just-in-time memento that provides

some style for a previously non-existent entity can be considered a generative transform. Taking an ID and generating a coherent collection of attributes that conforms to other constraints that might be present such as district, time of day, difficulty setting, or level of player.

14.2.4 Sorted Table

A container that keeps itself sorted, tunable for different circumstances including only sorting a small subset.

A table that is only ever read after sorting can attempt to keep track of changes rather than sort before use. If the table needs to stay sorted no matter what, it can generate indexes with which it reorders the table on transform. There is hardly ever a good reason to explicitly sort a table, and usually there are better ways to organise data than sorted. Binary searches don't work as well in practice as b-trees due to the cache-friendly operations.

14.2.5 Multi sorted table

A container that keeps multiple sorted representations of itself available, tunable for circumstances including sorting different subsets based on different criteria.

Much like the sorted table, but when a table has multiple queries all requiring different sorting algorithms, then it's better to give up and only use indirection via indexes according to what order each query requires. This is similar to how databases add indexes for each column that requests it.

14.2.6 Gatherer

A method by which large tasks can be split off into multiple parallel tasks and can then be reduced back into one container for further processing.

When you run concurrent processes, but you require the output to be presented in one table, there is a final stage of any transform that turns all the different outputs into one final output. You can reduce the amount of work and memory taken up by using the concatenate function and generate a table that is a conc-tree, or you can use a gather transform. A gather transform has as many inputs as you have parallel tasks leading into the final output table, and each input is a queue of data waiting to enter the final transform. As long as the data is not meant to be sorted, then the gatherer can run while the concurrent transforms are running, peeling the transformed rows out of the processes independent output queues, and dropping them into the final output table. This allows for completely lockless parallelism of many tasks that don't require the data to be in any particular order at the final stage.

If the gatherer is meant to produce the final table in the same order as the input table then you will need to pre-parse the data or have the gatherer use a more complex collating technique. When you pre-parse the data you can produce a count that can be used to initialise the output positions making the output table secure for the processing transforms to write directly into it, but if there is no way to pre-parse the data, then gatherer might need to co-operate with the task scheduler so that the processes are biased to return ealier work faster, making final collation a just in time process for the transfer to the next transform.

14.2.7 Tasker

When a container is very large, break off pieces and run transforms as concurrent tasks.

When you have a single large table, you can transform it with one transform, or many. Using a tasker to fake multiple independent tables allows for concurrent transforming as each transform can handle their specific part of the input table without having to consider any of the other parts in any way.

14.2.8 Dispatcher

Rather than switch which instructions to run based on data in the stream, prefer producing new tables that can each be transformed more precisely.

When operating on data, sometimes the transform is dependent on the data. Normally any enumerable types or booleans that imply a state that can change what transform is meant to run are hidden behind existence based processing tables, but sometimes it's good to store the bool or enum explicitly for size sake. When that is the case, it can be beneficial to pre-process a table and generate a job table, a table with a row for each transform that is meant to be run on the data.

A dispatcher takes on input table and emits into multiple tables based on the data. This allows for rapid execution of the tasks without the overhead of instruction cache misses due to changing what instructions are going to be run based on the data as all the instructions are collected into each of the job tables.

14.3 Existing design patterns

Most of the Object-oriented design patterns aren't necessary in data-oriented development, but explaining why can be useful for migrating from the object-oriented way of working. Even though design patterns are founded in the world of object-oriented development, some design patterns have mirrors in the data-oriented world. Sometimes this is because the functionality is incredibly simple (such as with the factory), or because the design pattern is emulated by basic functionality of the data-oriented technique (such as flyweight).

14.3.1 Abstract Factories

The requirement for abstract factories comes from needing to generate new objects based on an overall scheme. This means creation can now be dependent on two data values: the first is the scheme,

the second is the type requested for creation. This inherently object-oriented design problem stems from the desire to reuse code to drive object creation, but as C++ stops you from using objects by their features unless already declared in some way, it is hard to retroactively apply old creation techniques to new types. Normally we set up our classes by inheriting from an abstract base class so we can later provide new classes under new creation objects. The abstract factory is the eventual extension of allowing the creation function to be chosen at runtime based on the type of object required (which is statically checked) and the scheme or style of the object (which is dynamically calculated by the abstract factory), so choosing the right concrete factory object to create your new object.

When writing component entity systems, this is the equivalent of having components being inserted into an entity based on two dimensional data. The fact that a component of a certain type needs creating (the create call) is emulated by the row in the component create table, and the abstract factory calling into a concrete factory is emulated by the value causing the create call to be dispatched to an appropriate final factory table. Only once it has been dispatched to the right table does it get inserted into the final concrete component table.

This pattern is useful for multi-platform object-oriented development where you might have to otherwise refactor the internals of your object creators, but with a component driven approach, you can switch your output tables, transforms, or complete tree of transforms without losing consistency where it is necessary.

14.3.2 Builder

The builder pattern is all about wrapping up collections of reusable low-level parts of a complex transform in order to provide a tool kit for macro scale programming within a problem domain. The builder pattern is used when you want to be able to switch which tool-kits are used, thus providing a way of giving a high level program the ability to direct the creation of objects through the interface of a

collection of tools.

This can be seen to parallel the tool-kit of transforms you tend to build when developing with tables. With a data-oriented functional programming approach, we don't immediately call builders, but instead generate the necessary build instructions that can then dispatched to the correct parallel processes where possible before being reduced into the final data. Usually the builders are stateless transforms that convert the request to build into data outputs, or take a small amount of input data and build it into larger or translated output.

Once you work out that the builder pattern is mostly a rewording of procedural reuse with the addition of switching which procedures are called based on a state in a larger scope, it is easy to see why the pattern is applicable in all programming paradigms.

14.3.3 Factory Method

The factory method allows for data controlled choice of what is created. This is mirrored well in the component driven programming model, as all components are created via predictably called functions, but whether they are called is down to the data. Rather than a method, we use creation tables. Instances of entities and components are created when the system gets around to updating those tables. The creation request table contains the same arguments as the factory method, so the type and its arguments are used in that table update to cause an insert into the component or entity table. There is never any in-place creation, as there is never any need of it. Anything too complex to create as part of the output from a single transform deserves the attention of an explicit creation, and also will find it hard to mitigate the cost of creation out of the order of normal object creation, so will be less tempted to trash the instruction cache or derail the flow of the game for a special case like we find frequently in object-oriented development.

14.3.4 Prototype

The prototype pattern makes a lot of sense in an object-oriented code base, as the ability to generate more class instances from another class, only to modify it slightly before releasing it as finished almost gives you the flexibility to generate new classes at run-time, but any flexibility still has to be built into the classes before they can be used as the classes can only inherit from types existing at compile time. Some Object-oriented languages do let you generate new features at run-time but C++ doesn't allow for it without considerable coercion.

The existence-based-processing way of working stops you from creating a prototype as its existence would imply that it is in the game. There is no place to hide a prototype. But, there is also little call for it as creating a prototype is about creating from a plan of sorts, and because all the component creation can be controlled via read-only or just-in-time mementos, its very easy to make new classes with their components defined and specialised without resorting to hacks like keeping in memory a runtime template for a class you might need later. Unlike the object-oriented prototype, the component system memento does not have all the feature of the final component set, it only has the features of a creation data-type and thus fit's its purpose more purely. It cannot accidentally be used and damaged in the way prototypes can, for example, prototypes can be damaged by global update functions that accidentally modified the state. Because the only difference between a prototype and the runtime instance is whether the program knows that they are prototypes, it's hard to secure them against unwanted messages.

14.3.5 Singleton

There is no reason to think that a single instance of a piece of data has any place in a data-oriented development. If the data is definitely unique, then it deserves to be exposed as a system wide global. It is only when globals are used for temporary store that they become dangerous. Singletons are an inherently object-oriented solution to

a problem that only persists if you truly want to write platform ignorant code. Initialising and deinitialising managers in specific order for component managers, or for hardware or dependency sake can just as easily be managed explicitly, and is less prone to surprising someone. When you change the initialisation order accidentally by referencing a singleton, that reference won't be seen as changing the order of initialisation and it has been seen to cause quite difficult to debug when the effect of the change in order is subtle enough to not crash immediately.

14.3.6 Adapter

An adapter provides an API wrapper around an otherwise incompatible class. Classes have their APIs locked down after they are created. If two classes that need to work with each other come from third party software libraries, the likelihood of them being naturally compatible is low to non-existent. An adapter wraps the callee class in a caller friendly API. The adapter turns the caller class friendly API calls into callee class friendly API calls.

When writing programs data-oriented, most data is structs of arrays or tables. In data-oriented development there is no reason for an adapter, as in the first case, a transform can easily be made to take arguments of the specific arrays used, and in the second case, the transform would normally take two or more schema iterators that traversed the table providing arguments to and capturing outputs from the transform. In a sense, every table transform iterator is an adapter because it converts the explicit form of the table into the explicit form accepted by the transform.

Finally, data itself has no API, and thus the idea of an adapter is not appropriate as there is nothing to adapt from, so this relegates the idea to the concept of transforming the data so it can be provided to third party libraries. Preparing data for external processes is part of working with other code, and is one of the prices you pay for not having full control of the source code. The adapter pattern doesn't help this in any way, only provides a way of describing the inevitable.

14.3.7 Bridge

The bridge pattern is an object-oriented technique for providing encapsulation from implementation when the class's implementation is the functionality that derived classes use to get by in their world. For example, the platform specific code for an object can be privately owned by the parent class, and the concrete class can derive from that class. Then, it can use the implementation of it's parent class, but the implementation of the parent class member functions can be indirectly implemented differently depending on the platform. This way, the child class can call its parent's functions to do what it needs to do, while the parent class can call the implementation's functions which can be defined at runtime or compile time depending on what level of polymorphism is required.

Because all platform specific functionality is defined inside transforms and the way in which they are called, there is no reason to consider this pattern for platform independent reasons. All that is needed to benefit from the side effect of using this pattern is to recognise the boundary between platform specific transforms and platform independent transforms. For situations where the parent class implementation differs from instance to instance, you only need to notice that components can be different, yet still register with the same events. This means, any differing implementations required can be added as additional components which register into the same event tables as the existing components and replace those on creation as part of the different creation sequence.

Also, transforms that take data and do with it something platform specific would be written statically for each platform. This kind of object-oriented approach to everything can be what gives object-oriented development such a bad name when it comes to performance. There's not really any benefit to making your choice of implementation a runtime polymorphic call when the call can only ever be to one concrete method per platform as, at least in C++, there is not such thing as platform independent builds. For C#, it can be different, but the benefit of doing it is just the same, you

only benefit very slightly when it comes to readability in the code, but you lose on every call at runtime.

14.3.8 Composite

Unlike it's programming namesake, the composite pattern is not about an object being made out of multiple other classes by owning them in the implementation. The composite pattern is the idea that an object that is part of a system can be identified as a single object, or as a group of objects. Usually, a game entity references one renderable unit with a single AI instance, which is why this pattern can be useful. If an enemy unit class can represent a single unit, or a squad of units, then it becomes easier to manage commands in a real time strategy game, or organise AI path finding.

When you don't use objects, the level of detail of individual parts of entities are not so hard tied to each other as the entities are not objects. When you design your software around objects you get caught up thinking about each object as being indivisible, and thus the composite pattern lets you at least handle objects in groups. If you don't limit your data by tying it to objects, then the pattern is less useful as you can already manipulate the data by group in any way you require.

14.3.9 Decorator

The decorator acts much like adding more components to an entity. Their effect comes online when their existence marks their necessity. Decorators add features, which is exactly like the entity components with their ability to add attributes and functionality to a referenced entity.

14.3.10 Façade

Even though there are no tangled messes of interacting objects in the data-oriented approach, sometimes it might be better to channel

data through a simpler set of streams. This could be considered to offer the same smaller surface area that the façade offers.

14.3.11 Flyweight

The flyweight pattern offers a way to use very fine grain elements in the same way it uses larger constructs. In data-oriented development there is no need to differentiate between the two, as there is no inherent overhead in interfacing data in a data-oriented manner. The reason for a flyweight is similar to the reason for the bridge pattern, once you start thinking object-oriented and want to keep your whole codebase working like that, you can end up painting yourself into a corner about it and start adding objects to everything, even things that are too small or fragile for such a heavy handed, human centric way of seeing and manipulating the data.

14.3.12 Proxy

In a sense, the proxy pattern is much like the level of detail system in most games. You don't load final assets until you are near enough to need them. This topic was fully covered in chapter 4.

14.3.13 Chain Of Responsibility

The only problem with the chain of responsibility is that in most implementations it is inherently serial for all the possible recipients interested in responding to it. If we remove the possibility of the event being marked as handled, then the pattern is better handled through existence based processing than through object-oriented design, as existence-based-processing offers a way to subscribe at multiple points in the same entity with different components wanting to respond to different parts of a new event. If you want to enable such things as marking as handled, then it might be better to have a responsibility table for each message or event type with a dispatcher heading off the message before it hits the general message queue.

14.3.14 Command

This is realised fully when using tables to initiate your actions. All actions have to exist as a row in a table at some point, usually as a condition table row in the first place.

14.3.15 Interpreter

This design pattern doesn't seem very well suited to Object-oriented programming at all, though it is rooted in data driven flow control. The problem with the interpreter is it seems to be the design pattern for parsers, but doesn't add anything in itself.

14.3.16 Iterator

Although the table iterators that convert table schema into transform arguments are iterators, they don't fit into the design pattern because they are a concrete solution to iterating over tables. There is no scope for using them for arguments to functions or using them to erase elements and continue, they are stateful, but restricted to their one job. However, because there are no containers in table oriented solutions other than the tables, we can safely ignore true iterators as a design. In a stream based development, iterators are so common and pure, they don't really exist in the design pattern sense.

14.3.17 Mediator

The mediator can be thought of as a set up involving attaching event transforms to table updates. When a table entry updates, the callback can then inform the other related tables that they need to consider some new information. However, this data hiding and decoupling is not as necessary in data-oriented development as most solutions are considering all the data and thus don't need to be decoupled at this level.

14.3.18 Memento

Stashing a compressed state in a table is very useful when changing level of detail. Much was made of this in count lodding. Mementos also provide a mechanism for undoing changes, which can be essential to low memory usage network state prediction systems, so they can rollback update, and fast forward to the new predicted state.

14.3.19 Null Object

A null object is a pattern to stop unnecessary branching and checking when a fetch operation returns a class instance. If the instance was null, then returning the null object allows the calling code to continue on as if it had received a valid object. This is unnecessary in table driven processing scenarios as the transforms run over data. There is no chance for a transform to run into a null pointer as pointers are banned in general.

14.3.20 Observer

The publish and subscribe model presented in the table based event system is an observer pattern and should be used whenever data relies on some other data. The observer patter is high latency, but provides for callbacks, event handling, and job starting. In most games there will be an observer registry for the frame sync, the physics tick, the AI tick and for any IO completion. User defined callback tables will be everywhere, including how many finite state machines are implemented.

14.3.21 State

Data controlling the flow of the program is considered an anti-pattern in data-oriented development, so this design pattern is to be understood, and never used in the way it is presented. Instead, have your state be implicitly defined by the table in which your

entity resides. Using table existince to drive the transform that is called is the basis of finite state machine update, if not the alphabet response.

This design pattern is highly dangerous for any game that wants to run at high speed with a high entity count.

14.3.22 Strategy

14.3.23 Template Method

The template method can still be used but overriding the calls by removing the template callbacks and adding the specific new callbacks would be how you organise the equivalent of overriding the virtual calls.

14.3.24 Visitor

The visitor pattern would be fine if it were not for the implicit allowance for it to be stateful. If the visitor was not allowed to accumulate during it's structure traversal, then it would be an in-place transform. By not defining it as a stateless entity, but a stateful object that walks the container, it becomes an inherently serial, cache unfriendly container analyser. This pattern in particular is quoted by people who don't understand the fundamental requirements of a transform function. If you talk to an Object-Oriented programmer about taking a list of data and doing a transform on it, they will tend to mention it as the visitor patter because there is a tendency to fit solutions to problems with design patterns. The best way to define a visitor pattern is an object that changes state based on the incoming data. The final state is dependent on the order in which the data is given to the visitor, thus it is a serial operation. The best analogy to a transform is not compatible with a visitor because a transform pattern denies the possibility of accumulation.

14.4 Locking out anti-patterns

In addition to the design patterns, the elements of reusable object-oriented software, there are also the anti-patterns, the elements of miserable software. Sometimes a particularly bad way of doing something turns up way more often than it should, and when it does, it begins to get a name for itself. data-oriented development helps reduce these by both not being as complex to begin with because the problem domain is not dragged into a new language of objects, nor does it have objects themelves, which can cause multiple problems due to their sometimes coarse and sometimes too fine nature.

14.4.1 Abstraction Inversion

Abstraction inversion is when some internal functions of a class are kept hidden, but they are useful, so an external developer has to reinvent those internal functions as external functions by using only the public interface, which itself will probably be using the internal functions these new external functions are trying to replicate. Do this enough times with even the simplest of classes and you can explode the amount of work required to do a simple job.

As there is no inherent data or function hiding during data-oriented development, there can be no barrier to using the functions or the transforms that you want to use.

14.4.2 Acme Pattern / God Class

One giant class that runs the show. One very large data clump that is almost certainly a mess when it comes to organising the data by when it is accessed. Because there are no objects and we normalise our data, this anti-patterns becomes relatively difficult to experience. There is still the chance that someone might try to implement a nosql version of data-oriented development, but even if they do, most of the problems go away under struct of arrays. When it comes to the

problem of all the methods in one class, then we know we're fine as there are no such boundaries.

14.4.3 CObject / Cosmic Hierarchy

Almost the pure opposite of the God Class, the CObject does nothing, and is inherited whether directly or indirectly by everything. Many game engines have a concept of an entity or an object, but they generally don't keep it pure out of good practice. The threat of starting a cosmic hierarchy has reduced the chance of people making another CObject. The idea that everything comes from a base object class tends to encourage hard to read code. Code that divines the type of object before use suffers from overly long preamble in functions, and suffers from runtime type checking overhead. It's common knowledge that the preamble code - such things as "if type == foo" or checks for NULL - or the runtime type checking aren't overly expensive. This common knowledge has long outlived its lifespan though, and is provably a lie.

At the root of a lot of object-oriented entity systems lies a core class to which everything can be cast. This usually leads to lots of problematic code that costs cycles and can cause bugs by not correctly identifying itself.

In data-oriented development there is no Cosmic base class, even entities in an entity system only refer to entities. Each component will have a manager, and the idea of any component sharing some common core with another component goes so far away from the point of separating an entity into components that you should be hard pressed to find something that commits this anti-pattern.

14.4.4 Hidden Requirements

Sometimes a basic design change is required by a long standing requirement accidentally omitted from the original and all subsequent design documents. In object-oriented development, making a refactor at a late stage is very costly as object contracts can be hard to

adjust without causing a very large amount of related changes to all the code that touches the changed class.

Luckily, in data-oriented development, changing everything does not get much harder the longer the project continues because there are documented tools and techniques for making large scale schema changes in database technologies, and table based development can reap benefits there by literally implementing the same migrational tool-chain that is used in those circumstances.

14.4.5 Stovepipe

The more that bugs are fixed and corner cases are added, the harder it can be to refactor an existing Object-oriented codebase. This is not as true in data-oriented as towards the end of a project the difficultly in refactoring the design should be mitigated by the understanding and transparency of the whole program.

14.5 Game Development Patterns

The Game/Session/Level/Entity hierarchy still applies, and can be gathered accurately as part of the data flow analysis done during the early stages of design iteration. If you use tables for most of your data, then you will find that each layer has a set of tables.

Tick or update scheduling is normally necessary in order to maintain data scheduling, but as the data flow is paramount in data-oriented development, there is no need for a dedicated scheduler. In fact, the sequences of transforms replace the scheduler normally found in games by having a cascade of dependencies play out in the instance of a new frame being rendered. A frame buffer swap is an event that begins a new sequence of processes. In some games this could be the gathering of controller updates followed by physics and logic updates, finally culminating in an update of the renderable scene.

Double and Triple Buffering, which is normally the only way of getting near a good framerate, by processing multiple stages over multiple frames, may not need to happen so much as it does with object-oriented code as data-oriented code is much more parallelisable, so doesn't have the long critical path between input and rendering output. Removing multiple serial points can drastically reduce your overall gameplay feedback latency.

Chapter 15

What's wrong?

What's wrong with Object-Oriented Design? Where's the harm in it?

Over the years, games developers have taught themselves to write a style of C++ that is so unappealing to any current or future hardware, that the managed languages don't seem all that much slower in comparison. The current pattern of usage of C++ in games development is so appallingly mismatched to current console hardware that it is no wonder that an interpreted language is only in the region of 50% slower under normal use and sometimes faster[1] in their specialist areas. What is this strange language that has embedded itself into the minds of C++ games developers? What is it that makes the fashionable way of coding games one of the worst ways of making use of the machines we're targeting? Where, in essence, is the harm in game-development style object-oriented C++?

[1] http://keithlea.com/javabench/ tells the tale of the server JVM being faster than C++. There's some arguments against the results, but there's others that back it up. Read, make up your own mind.

15.1 The Harm

Virtuals don't cost much, but if you call them a lot it can add up.
aka - death by a thousand papercuts

The overhead of a virtual call is negligible under simple inspection. Compared to what you do inside a virtual call, the extra dereference required seems petty and very likely not to have any noticeable side effects other than dereference and the extra space taken up by the virtual table pointer. The extra dereference before getting the pointer to the function we want to call on this particular instance seems to be a trivial addition, but lets have a closer look at what is going on.

Firstly, we have a pointer to a class that has derived from a base class with virtual methods. This instantly adds the virtual table pointer as the implicit first data member of the class. Obviously, there is no way around this, it's in the language specification that the data layout be partially up to the compiler so it can implement such things as virtual methods by adding hidden members and generating new arrays of function pointers behind the scenes. If we try to call a virtual method on the class we have to read the first data member in order to access the right virtual table for calling. This requires loading from the address of the class into a register and adding an offset to the loaded value. Every virtual method call is a lookup into a table, so in the compiled code, all virtual calls are really function pointer array dereferences, which is where the offset comes in. It's the offset into the array of function pointers. Once the address of the real function pointer is generated, it is loaded into a register, then used to chance the program counter, via a branch instruction.

For multiple inheritance it is slightly more convoluted.

So let's count up the actual operations involved in this method call: first we have a load, then an add, then another load, then a branch. To almost all programmers this doesn't seem like a heavy cost to pay for runtime polymorphism. Four operations per call so that you can throw all your game entities into one array and loop through them updating, rendering, gathering collision state,

spawning off sound effects. This seems to be a good trade off, but it was only a good trade off when these particular instructions were cheap.

Two out of the four instructions are loads, which don't seem like they should cost much, but in actual fact, unless you hit the cache, a load takes around 600 cycles on modern consoles. The add is cheap, to modify the register value to address the correct function pointer, but the branch is not always cheap as it doesn't know where it's going until the second load completes. This could cause cache miss in the instructions. All in all, it's common to see around 1800 cycles wasted due to a single virtual call in any significantly large scale game. In 1800 cycles, the floating point unit alone could have finished naively calculating over fifty dot products, or up to 27 square roots. In the best case, the virtual table pointer will already be in memory, the object type the same as last time, so the function pointer address will be the same, and therefore the function pointer will be in cache too, and in that circumstance it's likely that the unconditional branch won't stall as the instructions are probably still in the cache too. But this best case is fairly uncommon.

The implementation of C++ doesn't like how we iterate over objects. The standard way of iterating over a set of heterogeneous objects is to literally do that, grab an iterator and call the virtual function on each object in turn. In normal game code, this will involve loading the virtual table pointer for each and every object. This causes a wait while loading the cache line, and cannot easily be avoided. Once the virtual table pointer is loaded, it can be used, with the constant offset (the index of the virtual method), to find the function pointer to call, however due to the size of virtual functions commonly found in games development, the table won't be in the cache. Naturally, this will cause another wait for load, and once this load has finished, we can only hope that the object is actually the same type as the previous element, otherwise we will have to wait some more for the instructions to load.

The reason virtual functions in games are large is that games developers have had it drilled into them that virtual functions are

okay, as long as you don't use them in tight loops, which invariably leads to them being used for more architectural considerations such as hierarchies of object types, or classes of solution helpers in tree like problem solving systems (such as path finding, or behaviour trees).

In C++, classes' virtual tables store function pointers by their class. The alternative is to have a virtual table for each function and switch function pointer on the type of the calling class. This works fine in practice and does save some of the overhead as the virtual table would be the same for all the calls in a single iteration of a group of objects. However, C++ was designed to allow for runtime linking to other libraries, libraries with new classes that may inherit from the existing codebase, and due to this, there had to be a way to allow a runtime linked class to add new virtual methods, and have them able to be called from the original running code. If C++ had gone with function oriented virtual tables, the language would have had to runtime patch the virtual tables whenever a new library was linked whether at link time for statically compiled additions, or at runtime for dynamically linked libraries. As it is, using a virtual table per class offers the same functionality but doesn't require any link time or runtime modification to the virtual tables as the tables are oriented by the classes, which by the language design are immutable during link time.

Combining the organisation of virtual tables and the order in which games tend to call methods, even when running through lists in a highly predictable manner, cache misses are commonplace. It's not just the implementation of classes that causes these cache misses, it's any time the instructions to run are controlled by data loaded. Games commonly implement scripting languages, and these languages are often interpreted and run on a virtual machine. However the virtual machine or JIT compiler is implemented, there is always an aspect of data controlling which instructions will be called next, and this causes branch misprediction. This is why, in general, interpreted languages are slower, they either run code based on loaded data in the case of bytecode interpreters, or they compile code just

in time, which though it creates faster code, causes cache misses and thrashing of its own.

When a developers implements and Object-oriented framework without using the built-in virtual functions, virtual tables and this pointers present in the C++ langauge, it doesn't reduce the chance of cache miss unless they use virtual tables by function rather than by class. But even when the developer has been especially careful, the very fact that they are doing Object-oriented programming with games developer access patterns, that of calling single functions on arrays of heterogeneous objects, they are still going to have the same cache misses as found when the virtual table has survived the call into the object. That is, the best they can hope for is one less cache miss per virtual call. That still leaves the opportunity for two cache misses, and an unconditional branch.

So, with all this apparent inefficiency, what makes games developers stick with Object-oriented coding practices? As games developers are frequently cited as a source of how the bleeding edge of computer software development is progressing, why have they not moved away wholesale from the problem and stopped using Object-oriented development practices all together?

15.2 Mapping the problem

Objects provide a better mapping from the real world description of the problem to the final code solution.

Object-oriented design when programming in games starts with thinking about the game design in terms of entities. Each entity in the game design is given a class, such as ship, player, bullet, or score. Each object maintains its own state, communicates with other objects through methods, and provides encapsulation so when the implementation of a particular entity changes, the other objects that use it or provide it with utility do not need to change. Games developers like abstraction as historically, they have had to write games

for not just one target platform, but usually at least two. In the past it was between console manufacturers, and now even games developers on the PC have to manage both Windows(tm) and MacOS(tm) platforms. The abstractions in the past were mostly hardware access abstractions, and naturally some gameplay abstractions as well, but as the game development industry matured, we found common forms of abstractions for areas such as physics, AI, and even player control. Finding these common abstractions allowed for third party libraries, and many of these use Object-oriented design as well. It's quite common for libraries to interact with the game through agents. These agent objects contain their own state data, whether hidden or publicly accessible, and provide functions by which they can be manipulated inside the constraints of the system that provided them. The game design inspired objects (such as ship, player, level) keep hold of agents and use them to find out what's going on in their world. A player object might use their physics agent to find out where they are going to be, and what is possible given where they are, and using their input agent, find out the player's intention, and in return providing some feedback to their physics agent about the forces, and information to their animation agent about what animations they need to play. A non-player character could mediate between the world physics agent and its AI agent. A car object could contain references to physics data for keeping it on the road, and also runtime modifiable rendering information can be chained to the collision system to show scratches and worse.

The entities in Object-oriented design are containers for data and a place to keep all the functions that manipulate that data. Don't confuse these entities with those of entity systems, as the entities in Object-oriented design are immutable over their lifetime. An Object-oriented entity does not change class during its lifetime in C++ because there is no process by which to reconstruct a class in place in the language. As can be expected, if you don't have the right tools for the job, a good workman works around it. Games developers don't change the type of their objects at runtime, instead they create new and destroy old in the case of a game entity that

needs this functionality.

For example, in a first person shooter, an object will be declared to represent the animating player mesh, but when the player dies, a clone would be made to represent the dead body as a rag doll. The animating player object is made invisible and moved to their next spawn point while the dead body object with its different set of virtual functions, and different data, remains where the player died so as to let the player watch their dead body. To achieve this sleight of hand, where the dead body object sits in as a replacement for the player once they are dead, copy constructors need to be defined. When the player is spawned back into the game, the player model will be made visible again, and if they wish to, the player can go and visit their dead clone. This works remarkably well, but it is a trick that would be unnecessary if the player could become a dead rag doll rather than spawn a clone of a different type. There is inherent danger in this too, that the cloning could have bugs, and cause other issues, and also that if the player dies but somehow is allowed to resurrect, then they have to find a way to convert the rag doll back into the animating player, and that is no simple feat.

Another example is in AI. The finite state machines that run most game AI maintain all the data necessary for all their potential states. If an AI has three states, { Idle, Making-a-stand, Fleeing-in-terror } then it has the data for all three states. If the Making-a-stand has a scared-points accumulator for accounting their fear, so that they can fight, but only up until they are too scared to continue, and the Fleeing-in-terror has a timer so that they will flee, but only for a certain time, then Idle will have these two unnecessary attributes as well. In this trivial example, the AI class has three data entries, { state, how-scared, flee-time }, and only one of these data entries is used by all three states. If the AI could change type when it transitioned from state to state, then it wouldn't even need the state member, as that functionality would be covered by the virtual table pointer. The AI would only allocate space for each of the state tracking members when in the appropriate state. The best we can do in C++ is to fake it by changing the virtual table pointer by

hand, or setting up a copy constructor for each possible transition.

This implementation detail does not cost much, can be worked around, but when it does, it bends if not breaks the standard way of doing Object-oriented development in C++.

Apart from immutable type, Object-oriented development also has a philosophical problem. Consider how humans perceive objects in real life. There is always a context to every observation. The humble table, when you look at it, you may see it to be a table with four legs, made of wood and modestly polished. If so, you will see it as being a brown colour, but you will also see the reflection of the light. You will see the grain, but when you think about what colour it is, you will think of it as being one colour. However, if you have the training of an artist, you will know that what you see is not what is actually there. There is no solid colour, and if you are looking at the table, you cannot see its precise shape, but only infer it. If you are inferring that it is brown by the average light colour entering your eye, then does it cease to be brown if you turn off the light? What about if there is too much light and all you can see is the reflection off the polished surface? If you look at its rectangular form from one of the long sides, then you will not see right angle corners, but instead a trapezium. We automatically classify objects when we see them, apply our prejudices to them and lock them down to make reasoning about them easier. This is why Object-oriented development is so appealing to us. However, what we find easy to consume as humans, is not optimal for a computer. When we think about game entities being objects, we think about them as wholes. But a computer has no concept of objects, and only sees objects as being badly organised data and functions organised for maximal cache abuse.

If you take another example from the table, consider the table to have legs about three feet long. That's a standard table. If the legs are only one foot long, it could be considered to be a coffee table. Short, but still usable as a place to throw the magazines and leave your cups. But when you get down to one inch long legs, it's no longer a table, but instead just a large piece of wood

with some stubs stuck on it. We can happily classify the same item but with different dimensions into three distinct classes of object. Table, coffee table, lump of wood with some little bits of wood on it. But, at what point does the lump of wood become a coffee table? Is it somewhere between 4 and 8 inch long legs? This is the same problem as presented about sand, when does it transition from grains of sand, to a pile of sand? How many grains are a pile, are a dune? The answer must be that there is no answer. The answer is also helpful in understanding how a computer thinks. It doesn't know the specific difference between our human classifications because to a certain degree even humans don't.

In most games engines, the Object-oriented approach leads to a lot of objects in very deep hierarchies. A common ancestor chain for an entity might be: PlayerEntity → CharacterEntity → MovingEntity → PhysicalEntity → Entity → Serialisable → ReferenceCounted → Base.

These deep hierarchies virtually guarantee you'll never have a cache hit when calling virtual methods, but they also cause a lot of pain when it comes to cross-cutting code, that is code that affects or is affected by concerns that are unrelated, or incongruous to the hierarchy. Consider a normal game with characters moving around a scene. In the scene you will have characters, the world, possibly some particle effects, lights, some static and some dynamic. In this scene, all these things need to be rendered, or used for rendering. The traditional approach is to use multiple inheritance or to make sure that there is a Renderable base class somewhere in every entity's inheritance chain. But what about entities that make noises? Do you add an audio emitter class as well? What about entities that are serialised vs those that are explicitly managed by the level? What about those that are so common that they need a different memory manager (such as the particles), or those that only optionally have to be rendered (like trash, flowers, or grass in the distance). Traditionally this has been solved by putting all the most common functionality into the core base class for everything in the game, with special exceptions for special circumstances, such

as when the level is animated, when a player character is in an intro or death screen, or is a boss character (who is special and deserves a little more code). These hacks are only necessary if you don't use multiple inheritance, but when you use multiple inheritance you then start to weave a web that could ultimately end up with virtual inheritance, and that can cause some serious performance overhead as it has to figure out a lot more than just where the virtual table is. The compromise almost always turns out to be some form of cosmic base class anti-pattern.

Object-oriented development is good at providing a human oriented representation of the problem in the source code, but bad at providing a machine representation of the solution. It is bad at providing a framework for creating an optimal solution, so the question remains: why are games developers still using Object-oriented techniques to develop games? It's possible its not about better design, but instead about make it easier to change the code. It's common knowledge that games developers are constantly changing code to match the natural evolution of the design of the game, right up until launch. Does Object-oriented development provide a good way of making maintenance and modification simpler or safer?

15.3 Internalised state

Encapsulation makes code more reusable. It's easier to modify the implementation without affecting the usage. Maintenance and refactoring become easy, quick, and safe.

The idea behind encapsulation is to provide a contract to the person using the code rather than providing a raw implementation. In theory, well written Object-oriented code that uses encapsulation is immune to damage caused by changing how an object manipulates its data. If all the code using the object complies with the contract and never directly uses any of the data members without going through accessor functions, then no matter what you change

about how the class fulfils that contract, there won't be any new bugs introduced by change. In theory, the object implementation can change in any way as long as the contract is not modified, but only extended. This is the open closed principle. A class should be open for extension, but closed for modification.

A contract is meant to provide some guarantees about how a complex system works. In practice, only unit testing can provide these guarantees.

Sometimes, programmers unwittingly rely on hidden features of objects' implementations. Sometimes the object they rely on has a bug that just so happens to fit their use case. If that bug is fixed, then the code using the object no longer works as expected. The use of the contract, though it was kept intact, has not helped the other piece of code to maintain working status across revisions. Instead it provided false hope that the returned values would not change. It doesn't even have to be a bug. Temporal couplings inside objects, or accidental or undocumented features that goes away in later revisions can also damage the code using the contract without breaking it.

One example would be the case where an object has a method that returns a list. If the internal representation is such that it always maintains a sorted list, and when the method is called it returns some subset of that list, it would be natural to assume that in most implementations of the method the subset would be sorted also. A concrete example could be a pickup manager that kept a list of pickups sorted by name. If the function returns all the pickup types that match a filter, then the caller could iterate the returned list until it found the pickup it wanted. To speed things up, it could early-out if it found a pickup with a name later than the item it was looking for, or it could do a binary search of the returned list. In both those cases, if the internal representation changed to something that wasn't ordered by name, then the code would no longer work. If the internal representation was changed so it was ordered by hash, then the early-out and binary search would be completely broken.

Another example of the contract being too little information, in

many linked list implementations, there is a decision made about whether to store the length of the list or not. The choice to store a count member will make multi-threaded access slower, but the choice not to store it will make finding the length of the list an O(n) operation. For situations where you only want to find out whether the list if empty, if the object contract only supplies a get_count() function, you cannot know for sure whether it would be cheaper to check if the count was greater than zero, or check if the being() and end() are the same.

Encapsulation only provides a way to hide bugs and cause assumptions in programmers. There is an old saying about assumptions, and encapsulation doesn't let you confirm or deny them unless you have access to the source code. If you have, and you need to look at it to find out what went wrong, then all that the encapsulation has done is add another layer to work around rather than add any functionality of its own.

15.4 Hierarchical design

Inheritance allows reuse of code by extension. Adding new features is simple.

Inheritance was seen as a major reason to use classes in C++ by games programmers. The obvious benefit was being able to inherit from multiple interfaces to gain attributes or agency in system objects such as physics, animation, and rendering. In the early days of C++ adoption, the hierarchies were shallow, not usually going much more than three layers deep, but later it became commonplace to find more than nine levels of ancestors in central classes such as that of the player, their vehicles, or the AI players. For example, in UnrealTournament, the minigun ammo object had this:

Miniammo → TournamentAmmo → Ammo → Pickup → Inventory → Actor → Object

Games developers use inheritance to provide a robust way to

implement polymorphism in games, where many game entities can be updated, rendered, or queried en-mass, without any hand coded checking of type. They also appreciate the reduced copy pasting, because inheriting from a class also adds functionality to a class. This early form of mix-ins was seen to reduce errors in coding as there were often times where bugs only existed because a programmer had fixed a bug in one place, but not all of them. Gradually, multiple inheritance faded into interfaces only, the practice of only inheriting from one real class, and any others had to be pure virtual interface classes as per the Java definition.

Although it seems like inheriting from class to extend its functionality is safe, there are many circumstances where classes don't quite behave as expected when methods are overridden. To extend a class, it is often necessary to read the source, not just of the class you're inheriting, but also the classes it inherits too. If a base class creates a pure virtual method, then it forces the child class to implement that method. If this was for a good reason, then that should be enforced, but you cannot enforce that every inheriting class implements this method, only the first instantiable class inheriting it. This can lead to obscure bugs where a new class sometimes acts or is treated like the class it is inheriting from.

Another pitfall of inheritance in C++ comes in the form of runtime versus compile time linking. A good example is default arguments on method calls, and badly understood overriding rules. What would you expect the output of the following program to be?

```
1  class A {
2    virtual void foo( int bar = 5 ) { cout << bar; }
3  };
4  class B : public A {
5    void foo( int bar = 7 ) { cout << bar * 2; }
6  };
7  int main( int argc, char *argv[] ) {
8    A *a = new B;
9    a->foo();
10   return 0;
11 }
```

Would you be surprised to find out it reported a value of 10? Some code relies on the compiled state, some on runtime. Adding new functionality to a class by extending it can quickly become a dangerous game as classes from two layers down can cause coupling side effects, throw exceptions (or worse, not throw an exception and quietly fail,) circumvent your changes, or possibly just make it impossible to implement your feature as they might already be taking up the namespace or have some other incompatibility with your plans, such as requiring a certain alignment or need to be in a certain bank of ram.

Inheritance does provide a clean way of implementing runtime polymorphism, but it's not the only way as we saw earlier. Adding a new feature by inheritance requires revisiting the base class, providing a default implementation, or a pure virtual, then providing implementations for all the classes that need to handle the new feature. Obviously this has required access to the base class, and possible touching of all child classes if the pure virtual route is taken, so even though the compiler can help you find all the places where the code needs to change, it has not made it significantly easier to change the code.

Using a type member instead of a virtual table pointer can give you the same runtime code linking, could be better for cache misses, and could be easier to add new features and reason about because it has less baggage when it comes to implementing those new features, provides a very simple way to mix and match capabilities compared to inheritance, and keeps the polymorphic code in one place. For example, in the fake virtual function go-forward, the class Car will step on the gas. In the class Person, it will set the direction vector. In the class UFO, it will also just set the direction vector. This sounds like a job for a switch statement fall through. In the fake virtual function re-fuel, the class Car and UFO will start a re-fuel timer and remain stationary while their fuelling up animatios play, whereas the Person class could just reduce their stamina-potion count and be instantly refuelled. Again, a switch statement with fall through provides all the runtime polymorphism you need, but you don't need

to multiple inherit in order to provide differing functionality on a per class per function level. Being able to pick what each method does in a class is not something that inheritance is good at, but it is something desirable, and non inheritance based polymorphism does allow it.

The original reason for using inheritance was that you would not need to revisit the base class, or change any of the existing code in order to extend and add functionality to the codebase, however, it is highly likely that you will at least need to view the base class implementation, and with changing specifications in games, it's also quite common to need changes at the base class level. Inheritance also inhibits certain types of analysis by locking people into thinking of objects as having IS-A relationships with the other object types in the game. A lot of flexibility is lost when a programmer is locked out of conceptualising objects as being combinations of features. Reducing multiple inheritance to interfaces, though helping to reduce the code complexity, has drawn a veil over the one good way of building up classes as compound objects. Although not a good solution in itself as it still abuses the cache, a switch on type seems to offer similar functionality to virtual tables without some of the associated baggage. So why put things in classes?

15.5 Divions of labour

Modular architecture for reduced coupling and better testing

The object-oriented paradigm is seen as another tool in the kit when it comes to ensuring quality of code. Strictly adhering to the open closed principle of always using accessors, methods, and inheritance to use or extend objects, programmers write significantly more modular code than they do if programming from a purely procedural perspective. This modularity separates each object's code into units. These units are collections of all the data and methods that act upon the data. It has been written about many times

that testing objects is simpler because each object can be tested in isolation.

However, Object-oriented design suffers from the problem of errors in communication. Objects are not systems, and systems need to be tested, and systems comprise of not only objects, but their inherent communication. The communication of objects is difficult to test, because in practice, it is hard to isolate the interactions between classes. Object-oriented development leads to an Object-oriented view of the system which makes it hard to isolate non-objects such as data transforms, communication, and temporal coupling.

Modular architecture is good because it limits the potential damage caused by changes, but just like encapsulation before, the contract to any module has to be unambiguous so as to reduce the chance of external reliance on unintended side effects of the implementation.

The reason Object-oriented modular approach doesn't work as well, is that the modules are defined by object boundary, not by a higher level concept. Good examples of modularity include stdio's FILE, the CRT's malloc/free, The NvTriStrip library's GenerateStrips. Each of these provide a solid, documented, narrow set of functions to access functionality that could otherwise be overwhelming and difficult to reason about.

Modularity in Object-oriented development can offer protection from other programmers who don't understand the code. An object's methods are often the instruction manual for an object in the eyes of a co-worker, so writing all the important manipulation methods in one block can give clues to anyone using the class. The modularity is important here because game development objects are regularly large, offering a lot of functionality spread across their many different aspects. Rather than find a way to address cross cutting concerns, game objects tend to fulfil all requirements rather than restrict themselves to their original design. Because of this bloating, the modular approach, that is, collecting methods by their concern in the source, can be beneficial to programmers coming at the object fresh. The obvious way to fix this would be to use a

paradigm that supports cross cutting concerns at a more fundamental level, but Object-oriented development is know to be inefficient at representing this in code.

If Object-oriented development doesn't increase modularity in such a way as it provides better results than explicitly modularising code, then what does it offer?

15.6 Reusable generic code

Faster development time through reuse of generic code

It is regarded as one of the holy grails of development to be able to consistently reduce development overhead by reusing old code. In order to stop wasting any of the investment in time and effort, it's been assumed that it will be possible to put together an application from existing code and only have to write some minor new features. The unfortunate truth is that any interesting new features you want to add will probably be incompatible with your old code and old way of laying out your data, and you will need to either rewrite the old code to allow for the new feature, or rewrite the old code to allow for the new data layout. If a software project can be built from existing solutions, objects that were invented to provide features for an old project, then it's probably not very complex. Any project for significant complexity includes hundreds if not thousands of special case objects that provide all particular needs of that project. For example, the vast majority of games will have a player class, but almost none share a common core set of attributes. Is there a world position member in a game of poker? Is there a hit point count member in the player of a racing game? Does the player have a gamer tag in a purely offline game? Having a generic class that can be reused doesn't make the game easier to create, all it does is move the specialisation into somewhere else. Some game toolkits do this by allowing script to extend the basic classes. Some game engines limit the gameplay to a certain genre and allow extension

away from that through data driven means. No-one has so far created a game API, because to do so, it would have to be so generic that it wouldn't provide anything more than what we already have with our languages we use for development.

Reuse, being hankered after by production, and thought of so highly by anyone without much experience in making games, has become an end in itself for many games developers. The pitfall of generics is a focus on keeping a class generic enough to be reused or re-purposed without thought as to why, or how. The first, the why, is a major stumbling block and needs to be taught out of developers as quickly as possible. Making something generic, for the sake of generality, is not a valid goal. It adds time to development without adding value. Some developers would cite this as short sighted, however, it is the how that deflates this argument. How do you generalise a class if you only use it in one place? The implementation of a class is testable only so far as it can be tested, and if you only use a class in one place, you can only test that it works in one situation. If you then generalise the class, yet don't have any other test cases than the first situation, then all you can test is that you didn't break the class when generalising it. So, if you cannot guarantee that the class works for other types or situations, all you have done by generalising the class is added more code for bugs to hide in. The resultant bugs are now hidden in code that works, possibly even tested, which means that any bugs introduced during this generalising have been stamped and approved.

Test driven development implicitly denies generic coding until the point where it is a good choice to do so. The only time when it is a good choice to move code to a more generic state, is when it reduces redundancy through reuse of common functionality.

Generic code has to fulfil more than just a basic set of features if it is to be used in many situations. If you write a templated array container, access to the array through the square bracket operators would be considered a basic feature, but you will also want to write iterators for it and possibly add an insert routine to take the headache out of shuffling the array up in memory. Little bugs

can creep in if you rewrite these functions whenever you need them, and linked lists are notorious for having bugs in quick and dirty implementations. To be fit for use by all users, any generic container should provide a full set of methods for manipulation, and the STL does that. There are hundreds of different functions to understand before you can be considered an STL-expert, and you have to be an STL-expert before you can be sure you're writing efficient code with the STL. There is a large amount of documentation available for the various implementations of the STL. Most of the implementations of the STL are very similar if not functionally the same. Even so, it can take some time for a programmer to become a valuable STL programmer due to this need to learn another langauge. The programmer has to learn a new language, the language of the STL, with its own nouns verbs and adjectives. To limit this, many games companies have a much reduced feature set reinterpretation of the STL that optionally provides better memory handling (because of the awkward hardware), more choice for the containers (so that you may choose a hash_map or trie, rather than just a map), or explicit implementations of simpler containers such as stack or singly linked lists and their intrusive brethren. These libraries are normally smaller in scope and are therefore easier to learn and hack than the STL variants, but they still need to be learnt and that takes some time. In the past this was a good compromise, but now the STL has extensive online documentation, there is no excuse not to use the STL except where memory or compilation time is concerned.

The takeaway from this however, is that generic code still needs to be learnt in order for the coder to be efficient, or not cause accidental performance bottlenecks. If you go with the STL, then at least you have a lot of documentation on your side. If your game company implements an amazingly complex template library, don't expect any coders to use it until they've had enough time to learn it, and that means that if you write generic code, expect people to not use it unless they come across it accidentally, or have been explicitly told to, as they won't know it's there, or won't trust it. In other words, starting out by writing generic code is a good way to write a

lot of code quickly without adding any value to your development.

Chapter 16

Looking at hardware

The first thing a good software engineer does when starting work on a new platform is read the contents listings in all the hardware manuals. The second thing is usually try to get hello world up and running. It's uncommon for a games development software engineer to decide it's a good idea to read all the documentation available. When they do, they will be reading them literally, and still probably not getting all the necessary information. When it comes to understanding hardware, there is the theoretical restrictions implied by the comments and data sheets in the manuals, but there is also the practical restrictions that can only be found through working with the hardware at an intimate level.

As most of the contemporary hardware is now API driven, with hardware manuals only being presented to the engineers responsible for graphics, audio, and media subsystems, it's tempting to start programming on a new piece of hardware without thinking about the hardware at all. Most programmers working on big games in big studios don't really know what's going on at the lower levels of their game engines they're working on, and to some extent that's probably good as it frees them up to write more gameplay code, but there comes a point in every developer's life when they have to bite

the bullet and find out why their code is slow. Some day, you're going to be five weeks from ship and need to claw back five frames a second on one level of the game that has been optimised in every other area other than yours. When that day comes, you'd better know why your code is slow, and to do that, you have to know what the hardware is doing when it's executing your code.

Some of the issues surrounding code performance are relevant to all hardware configurations. Some are only pertinent to configurations that have caches, or do write combining, or have branch prediction, but some hardware configurations have very special restrictions that can cause odd, but simple to fix performance glitches caused by decisions made during the chip's design process. These glitches are the gotchas of the hardware, and as such, need to be learnt in order to be avoided.

When it comes to the overall design of console CPUs, The XBox360 and PS3 are RISC based, low memory speed, multi-core machines, and these do have a set of considerations that remain somewhat misunderstood by mainstream game developers. Understanding how these machines differ from the desktop x86 machines that most programmers start their development life on, can be highly illuminating. The coming generation of consoles and other devices will change the hardware considerations again, but understanding that you do need to consider the hardware can sometimes only be learned by looking at historic data.

16.1 Sequential data

When you process your data in a sequence, you have a much higher chance of cache hit on reading in the data. Making all your calculations run from sequential data not only helps hardware with caches for reading, but also when using hardware that does write combining.

In theory, if you're reading one byte at a time to do something, then you can almost guarantee that the next byte will already be in

memory the next 127 times you look. For a list of floats, that works out as one memory load for 32 values. If you have an animation that has less than 32 keys per bone, then you can guarantee that you will only need to load as many cache-lines as you have bones in order to find all the key indexes into your arrays of transforms.

In practice, you will find that this only applies if the process you run doesn't load in lots of other data to help process your stream. That's not to say that trying to organise your data sequentially isn't important, but that it's just as important to ensure that the data being accessed is being accessed in patterns that allow the processors to leverage the benefits of that form. There is no point in making data sequential if all you are going to do is use it so slowly that the cache fills up between reads.

Sequential data is also easier to split among different processors as there is little to no chance of cache sharing. When your data is stored sequentially, rather than randomly, you know where in memory the data is, and so you can dispatch tasks to work on guaranteed unshared cache-lines. When multiple CPUs compete to write to a particular cache-line, they have to storing and loading to keep things consistent. If the data is randomly placed, such as when you allocate from a memory pool, or directly from the heap, you cannot be sure what order the data is in and can't even guarantee that you're not asking two different CPUs to work on the same cache-line of data.

The data-oriented approach, even when you don't use structs of arrays, still maintains that sequential data is better than random allocations. Not only is it good for the hardware, it's good for simplicity of code as it generally promotes transforms rather than object-oriented messaging.

16.2 Deep Pipes

CPUs perform instructions in pipelines. This is true of all processors, however, the number of stages differs wildly. For games developers, it's important to remember that it affects all the CPUs they work

on, from the current generation of consoles such as Sony's PS3 and Microsoft's Xbox360, but also to hand helds such as the Nintendo DS, the iPhone, and other devices.

Pipelines provide a way for CPUs to trade gains in speed for latency and branch penalties. A non-pipelined CPU finishes every instruction before it begins the next, however a pipelined CPU starts instructions and doesn't necessarily finish them until many cycles later. If you imagine a CPU as a factory, the idea is the equivalent of the production line, where each worker has one job, rather than each worker seeing and working on a product from start to finish. A CPU is better able to process more data faster this way because by increasing the latency, in well thought out programs, you only add a few cycles to any processing during prologue or epilogue. During the transform, latency can be mitigated by doing more work on non-related data while waiting for any dependencies. Because the CPUs have to do a lot less per cycle, the cycles take less time, which is what allows CPUs to get faster. What's happening is that it still takes just as long for a CPU to do an operation as it always has (give or take), but because the operation is split up into a lot of smaller stages, it is possible to do a lot more operations per second as all of the separate stages can operate in parallel, and any efficient code concentrates on doing this after all other optimisations have been made.

When pipelining, the CPU consists of a number of stages, firstly the fetch and decode stages, which in some hardware are the same stage, then an execute stage which does the actual calculation. This stage can take multiple cycles, but as long as the CPU has all the cycles covered by stages, it won't affect throughput. The CPU then finally stores the result in the last stage, dropping the value back into the output register or memory location.

With instructions having many stages, it can take many cycles for them to complete, but because only one part of the instruction is in use at each stage, a new instruction can be loaded as soon as the first instruction has got to the second stage. This pipe-lining allows us issue many more instructions than we could otherwise,

even though they might have high latency in themselves, and saves on transistor count as more transistors are being used at any one time. There is less waste. In the beginning, the main reason for this was that the circuits would take a certain amount of time to stabilise. Logic gates, in practice, don't immediately switch from one logic state to another. If you add in noise, resonance, and manufacturing error, you can begin to see that CPUs would have to wait quite a while between cycles, massively reducing the CPU frequency. This is why FPGAs cannot easily run at GHz speeds, they are arrays of flexible gate systems, which means that they suffer the most from stability problems, but amplified by the gates being located a long way from each other, in the sense that they are not right up next to each other like logic circuits are inside an inflexible ASIC like a production CPU.

Pipelines require that the instructions are ready. This can be problematic if the data the instruction is waiting on is not ready, or if the instruction is not loaded. If there is anything stopping the instruction from being issued it can cause a stall, or in the case of branching causing the instructions to be run, then trashed as the pipe-line is flushed ready to begin processing the correct branch. If the instruction pointer is determined by some data value, it can mean a long wait while the next instruction is loaded from main memory. If the next instruction is based on a conditional branch, then the branch has to wait on the condition, thus causing a number of instructions to begin processing when the branch could invalidate all the work done so far. As well as instructions needing to be nearby, the registers must already be populated, otherwise something will have to wait.

16.3 Microcode: virtually function calls.

To get around the limitations implicit in trying to increase throughput, some instructions on the RISC chips aren't really there. Instead, these virtual instructions are like function calls, calls to macros

that run a sequence of instructions. These instructions are said to be micro-coded, and in order to run, they often need to commandeer the CPU for their entire duration to maintain atomicity. Some functions are micro-coded due to their infrequent use or relative cost to implement as an intrinsic instruction, some because of the spec, and some because they don't fit well with the pipe-lined model of execution. In all of these cases, a micro-coded instruction causes a gap, called a bubble, in the pipeline, and that's wasted execution time. In almost all cases, these microcoded instructions can be avoided, sometimes by changing command line parameters (ref Cell Performance), sometimes by adjusting how you solve a problem (ref 1¡¡n hack), and sometimes by changing the problem completely. (sqrt considered harmful)

16.4 Single Instruction Multiple Data

There are no current generation consoles that don't have SIMD of some sort. All hardware now has some kind of vector unit, and to some extent, as long as you work within your boundaries, even hardware that doesn't have SIMD instructions, such as embedded micro controllers, can operate on multiple data. The idea behind SIMD is simple: issue one command, and manipulate multiple pieces of data in the same way at the same time. The most commonly referenced implementation of this is the vector units inherent in all current generation hardware. The AlitVec instructions on PPC and the SPU instruction set contain many instructions that operate on multiple pieces of data at the same time, sometimes doing asymetric operations such as rotating, splatting, or reconfiguring the vectors. On older machines or simple machines, the explicit instructions may not exist, but in the world of bitwise logic, we've always had some SIMD instructions hanging around as all the bitwise ops run over multiple elements in a bit field of whatever native word length. Consider some of the winners of the quickest bit counting routines. My favourite is the purely SIMD style bit counter given here:

```
uint32_t CountBits( uint32_t in ) {
  v = v - ((v >> 1) & 0x55555555);                    // reuse
       input as temporary
  v = (v & 0x33333333) + ((v >> 2) & 0x33333333);     // temp
  c = ((v + (v >> 4) & 0xF0F0F0F) * 0x1010101) >> 24; // count
  return c;
}
```

So, look to your types, and see if you can add a bit of SIMD to your development without even breaking out the vector instrinsics.

16.5 Predictable instructions

The biggest crime to commit in a deeply pipelined core is to tell it to do loads of instructions, then once it's almost done, change your mind and start on something completely different. This heinous crime is all too common, with control flow instructions doing just that when they're hard to predict, or impossible to predict in the case of entirely random data, or where the data pattern is known, but the architecture doesn't support branch predictions.

Chapter 17

Future

The future of hardware is almost certainly increasingly parallel processing with all the related problems that entails; from power consumption, through deserialising our processing, to distribution of jobs across more than just a few local cores.

Supercomputers have been vector machines for decades, and any advances in graphics cards comes with an increase in the number of cores that run transforms. Mobile phones have 4 cores and even some microcontrollers have multi-core layouts[1] Stream processing hardware is showing to be more capable of increasing throughput per generation than the general purpose computing architecture employed in the design of CPUs and micro-controllers. It's shaping up that the coming generations of hardware are going to be a battles of the parallel processing beasts[2], larger core counts and more numerous pipes. Memory always seems to be at a premium, so again, we'll probably be trying to push next gen games out without having enough space in which to work. One area that has not been pushed

[1] Parallax Propeller is a really interesting microcontroller that's been out for some time, but provides a unique perspective on parallel computing

[2] Parallela was a successful 2012 Kickstarter campaign for Adapteva, in which their roadmap points to 1024 core boards being produced in 2014

around much has been offloading processing outside the local machine. Even though there has been plenty of work done with very high latency architectures such as Beowulf clusters or grid computing, the idea that they could be useful for games is still in its infancy due to the virtually real-time nature of processing for games. It could be a while off, but we may see it come one day, and it's better not to let it sneak up on us. Once we need it, we will need a language that handles offloading to external processing like any other threaded task. Erlang is one such language. Erlang is functional (with a tendency for immutable state which helps very much with concurrency) and is highly process oriented with a very simple migration path from single-machine multi-threaded to remote processing or grid compute style distributed processing and a large number of options for fault tolerance. Node-js offers some of the same parallelism and a much shorter learning time as it is based on Javascript. Functional languages would seem to dominate, but OpenCL isn't purely functional, and C++AMP only requires you consider how to amplify your code with parallelism, which might not be enough for how many cores we really end up. Whatever language we do end up using to leverage the power of parallel processing, once it's ubiquitous, we'd better be sure we're not still thinking serial.

But parallel processing isn't the only thing on the horizon. Along with many cores comes the beginnings of a new issue that we're only starting to see traces of outside supercomputing. The issue of power per watt. In mobile gaming, though we're striving to make a game that works, one thing that developers aren't regularly doing that will affect sales in the years to come, is keeping the power consumption down on purpose. The mobile device users will put up with us eking out the last of the performance for only so long. There is a bigger thing at stake than being the best graphical performance and fastest AI code, there is also the need to have our applications be small enough to be kept on the device, and also not eat up enough battery that the users drop the application like a hot potato. Data-oriented design can address this issue, but only if we add it to the list of considerations when developing a game. As with parallelism,

the language we use impacts the possibilities. If we continue to move towards more high level languages such as C# and Java, then we will need smarter compilers to reduce the overhead, but if we develop in a language made to support tasking, such as the process kernels in OpenCL or the lightweight threads of Erlang, then we may find new hardware changes to match the language much the same way C++ and object-oriented design changed the way CPUs were designed.

17.1 Parallel processing

In short, parallel processing increases the number of things done at the same time. Towards an infinite number of CPUs the amount of processing available is infinite, but the amount of processing you can use is limited by two factors.

The first limiting factor is the amount of work that needs to be done. If you have a perfectly parallel algorithms, then you are limited by the number of units of work you have to do.

For example, if a graphics card had an infinite number of cores (and we'll keep using infinite as it is the only good measure for future values of N), then the maximum number of cores that it can use to render a single polygon would be measured in how many tasks there are to do. If the polygon was fully covering a standard 1080P HD screen with 1920 x 1080 pixels, then the highest feasible number of cores to throw at the task of pixel shading the poly to the back buffer would be 2,073,600, and thus the highest throughput you can possibly get from the machine would rely on how fast one single core could transform the request to render, into a final change of value at the pixel. This example is slightly broken because the time taken to set up that many cores in a normal rendering setup would probably cost more time than just splitting the task into larger chunks, and if you use MSAA you can call on even more cores, but eventually you'll even run out of samples to do. The point being, if there are not enough jobs, then the cores will be under utilised.

A lack of jobs is hard to find in a graphics card, which is part of

the reason they have been growing in power so rapidly over the last few years (2008-2013), and show no signs of significantly slowing down in their GFLOPS growth. There are always more pixels to render, always more vertices to transform, always more textures to lookup. The bottleneck in a graphics card is still the number of cores, and that's why it's relatively easy to increase the power of them compared to general purpose chips like CPUs.

In addition to the amount of jobs the cores have to do, the relative similarity of the jobs makes it easier to move towards a job-board style approach to job dispatch. In every parallel architecture so far, the instructions to run per compute core are decided specifically per core. In the future, an implicit job system may make a difference by instigating the concept of a public read-only job spec, and having set up the compute cores to look out for jobs and apply their own variant information on them. A good example might be that a graphics card may set all the compute cores to run the same algorithm, but given some constants about where they should get their data such as offsets into the stream and different output rectangles of the display.

This second limiting factor is dependency. We touched on it with the argument for pixel shaders, the set up time is a dependency. The number of cores able to be utilised is serial up until the first step of set up is finished, that is, the call to render primitive. Once the render call is done, we can bifurcate our way to parallelism, but even then, we're wasting a lot of power because of dependence on previous steps. One way to avoid this in hardware is have many cores wait for tasks to do by assuming they will be in charge of some particular part of the transform. We see it in SIMD processing where the CPU issues one instruction to multiply a vector, and each sub core carries out its own multiply, knowing that even though it was told that a multiply for the whole vector was issued, it could be sure that it was the core involved in multiplying element n of that vector. This is very useful and possibly why future optimisations of hardware can work towards a runtime configurable hardware layout that runs specified computations on demand, but without explicit instruction. Instead, as soon as any data enters into the transformation arena, it begins

the configured task without asking for, or needing information on how to interpret the data. This type of stream processing may be best suited to runtime configurable hardware.

Ahmdal's Law is based on two constants, the time that a process must be serial and the time that a process can be parallel. In the world of infinite core computing we must continually strive for the lowest possible serial latency in our development. That means we must find out what is critical, what can be done without prior information, what can be done in preparation, what can be not done until the very last moment. All these elements of processing add up to a full product, and without considering what the output data is and how we get there, we could continually return to the state where we are optimising for code, and not for what is of ultimate importance, the experience due to realisation of output data in a timely fashion.

17.2 Distributed computing

When you think of distributed computing, normally you think of farms of computers spread over multiple locations, running long lifetime processes on massive amounts of data. You think of on demand load balancing systems providing more compute where necessary on the grid. All this talk of large scale is just another step out from our CPU cores. Another layer that we can move to when the possibility presents itself. One day, maybe, we will have invisible grid computing for the consumer. The idea of spreading the compute load out from the physical device was first presented commercially with the CellBE, the idea that adding more compute to an existing hardware instance could be done without explicitly linking hardware through a proprietary interface. There were articles talking of how your TV could increase the quality of your gaming experience by allowing the console to offload some of the processing onto the TVs processor. This has not come to pass, but the common core, the idea of doing more work by adding more machines, that has not

gone away and is in common use in most development companies, whether by use of the free to use build accelerators such as the one included in the Sony developer tools, or proprietary software such as Xoreax's IncrediBuild. Adding more compute to processes that can safely be very high latency is definitely one way of using grid engines, but another alternative is to use grid engine's to provide answers to questions not yet asked. If you know that a type of question is common during your processing, then offloading the question process so it processes all known variants of the question means you can get back the answer to all the possible questions before it's even asked.

For example, If you know that your scene is going to animate, and you want to do a ray cast from some entities to other entities, but don't know which, you can task the grid engine with advancing the animation and doing all the ray casts. Then, once you are ready, read from the ray casts that you decided that you did need after whatever calculations you needed to do. That's a lot of wasted processing power, but it reduces latency. When you're thinking about the future, it is only sensible to think about latency as the future contains an infinite number of processing cores.

Using this one example, you can ramp it back to current generation hardware by offloading processes to begin large scale ray casting, and during the main thread calculation stages, cull tasks from the thread that is managing the ray casts. This means that you can get started on tasks, but only waste cycles when you don't know enough to not.

With very high speed network adapters, very high bandwidth network connections to the internet, grid computing inside games might become more commonplace than expected. What gameplay elements this amount of processing power can open up is beyond our vision, but once it is here, we can expect remote processing through something akin to stored procedures to become a staple part of the game developer tool-kit.

17.3 Runtime hardware configuration

We cannot know what the future brings, and some assumptions we have about CPUs might be broken. One such assumption might be that our hardware is fixed in one form once it has been fabricated. Both field programmable gate arrays and complex programmable logic arrays allow for change during their lifetime, and some announcements have been made about mainstream hardware adding FPGAs to their arsenal[3] which may open the door to FPGA add-on cards much the same way we saw the take-off of graphics cards once they got a foot in the door.

FPGAs present a new and interesting problem to programmers, specifically to the programmers that may be inventing new plans for others to consume. One way this could pan out is by re-orienting the mindset of the average programmer into that of a flow based or stream processing developer. Being able to take game data and manipulate it with highly efficient, and low latency modules dynamically loaded onto FPGAs could be the final nail in the coffin for any language that links code with data.

Approaches to development similar to that imposed on us by the shader model and the process of getting data into the right layout for the shaders might have been good training, readying us for the coming days of compute power only really being available if we order it for later. With dynamic partial reconfiguration, we would have the same benefits seen in being able to switch shaders mid render. That is, the ability to utilise an FPGA much smaller than would be necessary if we were to attempt to cram all the potential processing modules onto it at once. We've been here before. This extremely high latency before being able to compute, followed by very fast processing once the computational framework is ready, is very similar to programming with shaders, so much so that we might not take that long bringing our existing engines up to speed.

[3]One really good reason for an FPGA is to allow for future proofing of output connectivity. FPGAs can be used to test out new standards such as a new HDMI specification or a completely different encoding

The CellBE was an attempt at this way of working, but was overlooked by many as just a strange piece of hardware. The core, an underpowered CPU that would look after feeding all the high power SPUs was assumed to be a general purpose CPU. This was an unfortunate side effect of too many years developing for out-of-order CPUs, and a deep investment in random memory access programming methods such as object-oriented design and interpreted languages. We don't know what other CPUs may come along in the future, but we can attempt to use a data-oriented approach and not try to make a CPU work the way we want to work, but work with it to make the best out of what we have.

17.4 Data centric development in hardware design

Once software solutions concentrate on transforming data, what changes can we expect from hardware vendors? Is that a silly question? Would a change of programming paradigm really affect the people who create hardware?

What should we promote and demote in order to get the most from our transistors?

move away from serial thinking more functional programming, where there are no side-effects, leads to solutions for infinite numbers of cores

Whatever the future brings with respect to hardware and processing configurations, there are certain assumptions we can make.