3D game engines and design patterns

A case study of the Crystal Space game-engine

A report in CD5130

Siri Sjöqvist
sst01001@student.mdh.se

Erik Balgård
ebd00001@student.mdh.se

24 may 2006

Teacher: Martin Skogevall
Email: martin.skogevall@mdh.se
Department of Computer Science and Electronics at mälardalens högskola

Abstract

This report is intended as a case-study of different design-patterns used in 3D-game engines. It contains a general overview of game engine components and a case study of how the design patters laid forth in the famous book "Design Patterns: Elements of Reusable Object-Oriented Software" used in the Crystal Space game engine. Crystal Space is an open source General 3D development kit written in c++. Crystal Space is one of the largest open source 3D game engines and have been used as a base for several commercial products such as Plane-shift(mmorpg) and 'Bone Artz' who is in the making of "Bonez Adventures"(an interactive 3d adventure game).
The question we ask is "How is this game-engine constructed and more importantly what design-patterns were used in it's construction?". We are also interested in why they made those specific choices in patterns and what advantages or drawbacks they have experienced as a result of these choices. The primary reason behind the making of this study is to illustrate the practical uses of design patterns in our society and to give real examples of consequences involved when patterns.

Contents

1 Introduction

In this report we have studied game engines as a whole and then concentrated on a specific game engine called Crystal Space for a case study. What we want to illustrate is the use of design patterns in our society today and try to explain for those interested what they are and how they are used in our daily lives. We have tried to keep the language quite simple and have also tried to explain different terms and parts of the game engines in a way that even someone with not so much experience in this field would be able to understand and make use of. One reason that we chose Crystal Space engine for our little project was because it is the largest and most complete open source engine. It being open source was particularly important as this gave us a much larger chance of seeing the inner workings of the engine. Since it is open source it is also in a constant state of development which enables us to have a dialog with the developers.

2 Game Engines

2.1 Game Engines

2.1.1 What is a game engine?

In short, a game engine is the heart and core in any 3D game. It is the thing that takes care of all the behind the scenes calculations and functions, everything that is not in fact the actual game, which is the idea, the script and the artworks.

A game engine is, opposed to what most ordinary game-users might believe, not the actual game. It is the heart and the brain. The thing that makes your game work, let you move your character, allows you to see, hear and interact with other things and personas. You could think about it as the little engine that could. The core around which you as a developer would build your game, your world or what other product you wish to create. The difference between the game and the game engine would be that the game engine would just include code and functions to make the game-development easier and more simple. Picture it like this: If you have a game where you are a small figure that have to go through a maze and rescue the imprisoned Princess(tm) on the other side, the Game in this sense would be the idea of what you should do and what will happen when you do it, the characters included in this part and all necessary data for this to happen. Then mostly thinking about all that you can see, change or touch such as high-scores, weapons, menus, monsters, characters, background and so on so forth. The script if you will.
The game engine in this scenario would be the hidden mechanics behind the scenes, the thing that makes it possible for you to steer your character through the maze with joystick or keyboard, the thing that would render the scene which you see to make it look real with that ray of sunshine mirroring in the puddles of water, the thing that checks that you can't go straight through visible items on the screen, the thing that calculates the positioning of every item in the scene and how big or small in perspective it will be. So basically,

the idea, script and artworks are the Game, while the mechanics to show, use and control those are the Game Engine.

The game engine is not really consisting of just one big chunk of code that glues all parts of a game together but more usually it is different bits and pieces connected into one working unit. One could say that the game engine includes all those functions and operations that is general for all games, something that most or all games share. Those bits could be viewing the actual game, translate the game-functions into understandable commands for your working platform, calculate an item's position, calculate the layering of a screen-picture so that you as a user would believe it's real, or at least feel it's realistic.

2.1.2 The different parts of a game-engine

A typical game engine would include three big parts, the rendering engine, the physical engine and then the scene graph. Of course, any of those three units or engines are possible to change for another of its kind as different engines are good at different things.

The rendering engine

The rendering engine or the 'Renderer' is the one that will calculate how to actually view scenes for the user and how to store this information in the memory.
So take for example a 3D-figure. For this you would need information about how it is built, how it should look and other information such as 'will it reflect light?', ' does it glow in the dark?', 'does it cast shadows?'. The ground concept of 3D is that it should be viewable from more than one angle. A one-angle or one-side viewable object would be 2D. Following this line of thought we can easily conclude that the 3D-figure needs something more. So how is it made? A 3D-object is built up by points (called vertices) and polygons, vertices that relates to other vertices in it's surrounding and so creating an object. The most simple form or 3D-object would be a cube. A cube has 8 points (vertices), one in each corner and the relation between those would be the edges of the cube. The 6 sides of the cube would be the surfaces and in turn each of those surfaces is divided into two polygons (triangular shapes). So the render would store this shape as 8 points and some other information how to draw it such as its texture, material, value of reflection, transparency and other needed values. The render's work would then be to draw all the shapes and calculate how light would fall on each and every item in a scene and how they would interact with each other (like a sunray falling on a window or water-surface would make the light bounce and light up something else) where the shadows would fall considering the light-sources of the current scene and do this creating maximal effect and reality using the least of your memory and cpu-power. Some of this procedure is included in something called culling. If you have a first-shooter game for example, and a huge 3D-world to run around in, you still just have the one shooter-view and you won't be able to see the whole world. To make the game go faster and keep the frame-rate up, you only draw out the points and polygons that are visible from your current view, not the whole world, because no graphic-card would be able to handle this.

The physical engine

The physical engine is the one that probably has the most down to earth functions of the game engine. This is the part that handles collision detection & networking, plays sounds & animations and act as an artificial intelligence.

Collision detection

As it is normal nowadays to create games using object oriented languages most of the items in a game are objects of one or the other kind. Though even if it is not built up in this particular way you need to be able to check that item A is not colliding with item B, or rather, you need to check if they Do. The collision detection are methods that check if any item is colliding with another and then let you handle the result as you seem fit. For example you would need to check the collision if, in a shooter game, person A fires a Bullet (which is an object in itself) at person B (terrible fiend) to see if person A has hit person B with the Bullet. Then there can be options about what to do if Bullet miss person B and collides/hits the wall behind him, shall it ricochet or shall it collide/fall down on the floor? This would be an example of a game mimicking reality closely and most shooter games cheat in this area to actually lower the amount of collision detections they need to do. Collision detection is so a much bigger part of a game than you might think at a first glance but luckily there are not too many different ways to actually do it. Or at least, once you know what kind of collision detection you want or already have it's not too hard to build up the rest around it.

Sound

The sound feature of this physics-engine is what would store and play your sounds and with the games of today with their vast 3D-worlds and realistic sounds, it can become a quite dirty business. The sound is also a very important part of the games as it would heighten the impression and the atmosphere of the game. What this part does is to collect and store your sounds and tell your sound-card how to play them. Sound is, in a way, quite like Light. Just as a beam of light can bounce against items and walls so can the sound and in newer games we have become quite accustomed to that as everything Else is so realistic, so should the sound be. This means that you for example, will hear the sound a bit muffled when you are standing on one side of a wall and the one speaking stand on the other, or that the sound should be a little bit tweaked if the one you're listening to stand on the other side of a pillar. All this takes some calculation and monitoring. What basically needs to be done is to first decide what sounds you can hear, then how you hear them depending on where you are (different materials and environments makes sounds sound differently), how many sounds you'd hear and then last of all how to play those. The mixing of the sound is in most cases done in the sound-card but based on a history of changing standards, some game-engines have taken this step as well. With today's standards neither making sounds or storing them takes too much effort, there are tools for everything, but one feature of the sound-part of physics engine would be compressing and dec0ompressing sounds using your cpu. In the beginning mostly wav and midi files were used but today there are other formats such as mp3 and aac, which has a higher compression/decompression rate which makes it take less space and less effort for your computer.

Scripting

Scripting is another part of the physics-engine. This is a part that in much tells other parts what to do. For example you might be in an alley, scary music playing and suddenly you catch a glimpse of something bright, a bottle falls to the ground and There the script tells the software to load that awesome movie-clip, or sprout out 5 enemies, or some other action that is pre-decided for a certain spot or time in your game. You could call it a huge event-handler, something that reacts on triggers made by your character. A trigger can then be you open a door, you walk into a certain place, you beat a huge monster. Then again, entering a boss' lair will make the script play an eventual movie or animated clip and start the special scary music, load the boss itself and take other necessary actions. Scripts are pieces of code with lists of commands that happens when triggered. You could say the script for popping three monsters when you enter a deserted house and sprouting 5 monsters in the alley would basically be the same, there would just be some changed values as to location, amounts and id of creatures. It might not sound as something very hard but coding those yourself from scratch can be very time-consuming and so expensive, not to mention that it can be hard to stop. Scripts can be allowed to handle quite a lot of things, all from playing sounds to logging the character's life in the game and so decide events based on this to happen. An easy script could for example look something like 'if character A enter House through door B, then C numbers of guards come running and movie-clip D starts to play'.

Animation

Animation is another important aspect. Having beautiful models does not count for much if they move poorly. The animation in 3D computer games come in two varieties: mesh based animation and skeletal animation. In mesh based animation the animator have to specify the location of each point of every polygon of the model for each frame in the animation. This is the old way of doing things and often not only took a long time but it also was hard to make lifelike animations this way. Then along came skeletal animations. In skeletal animation each model is given a number of bones (just like a real skeleton) and when you manipulate one of them the computer calculates how the surrounding bones are affected. For example if you have a humanoid model which reaches for an object you only need to move the hand and the computer calculates how the arm and upper body needs to move in order to reach the desired object. This way not only saves time in the animation process, it also gives smoother, more lifelike animations.

Networking

As games have progressed the importance of multi-player capabilities using a network connection have increased to a point where it can be seen as a controversial choice by a game developer to not include it in their game. This has lead to a large number of solutions to common problems encountered when trying to develop an efficient network protocol for game use. There are two protocols that are used by almost all games today, they are UDP and TCP. TCP ensures that all packages sent arrive on the receiving end undamaged and in the order they were sent. While this sounds very good it requires the sender to resend a package if it was damaged on the way. TCP also hold packages that arrive out of order at the receiving end and wait for previous packages in order to be able to deliver them in the same order as they were sent. UDP is in a way the complete opposite of TCP. Were TCP is cautious and makes sure every package arrives correctly and takes the time to do resends if something goes wrong UDP just sends the packages and hope everything works out. UDP does not do any type of error checking or delivery verification. This may seem reckless but in today's high-speed, reliable networks the probability of an error occurring is small enough that the speedy delivery gained from using UDP outweighs the potential problems of corrupt packets. Networking: When it comes to different types of games and their network requirements most games can be placed somewhere on a scale with turn based games at one end and action games at the other. Turn-Based games place little importance on the latency of their communication but they instead want to make sure the packets arrive correctly. Because of this they often use TCP as their network layer protocol. Action games on the other hand often rely on split second decisions and should always use UDP to ensure low latency. These are the extreme points where the choice of network protocol is obvious, most games fall somewhere in between and then it's up to the game-designer to weigh the pros and cons of each model and make the best compromise.

Artificial Inteligence

The artificial Intelligence in a game is what makes your opponents or co-players seem real. For example in a shooter-game you have opponents that tries to kill you. A simple AI would let the opponent check where he is, where you are and then aim the gun at you and shoot which would mirror a very low amount of intelligence. In a more advanced AI he would act a bit different. The calculations behind his movements are derived from his position, your position, calculations where he should go to get to a place where he can shoot you and you can't get to him, the calculated path that he needs to take to avoid most damage and also, some estimations to your own actions and what they could possibly be at a given moment. This has been a part of games for quite a long time but the depth of the intelligence has changed from game to game. Having a game where you should mimic soldiers that try to advance on an enemy base, the AI would need to calculate movements of more than one, make it realistic and also keep in mind to move the soldiers in comparison to each other. It would have to take into account that if soldier A is attacked, would it be sensible of soldier B to go the same way and would it be a good idea to help soldier A?

The scene graph

The scene graph is the part of the engine that will help you keep track of things in your world, or rather arrange them in an order which is easy to view, use or transform if needed. The scene graph is built on an object oriented structure normally arranged in a tree or graph structure.

Tree-structure meaning one root-node(parent) with N-children nodes. The difference here between the two is that meanwhile a tree-structure basically just spreads out the graph, it can also gather together again. Needless to say the scene-graph can be used to many things and have not only been used for games but for other vector-based editing applications such as Adobe Illustrator or Corel Draw. Though, back to the structuring. The scene graph would handle your scenes and worlds as nodes in the tree, where a node most commonly can have only one single parent. Which would mean that any change or transformation made to a parent would be recognized by it's children. To give an example, you might have a House-node, then all the rooms inside the House will be its children, and yet again, objects inside the rooms such as furniture and even people will be the children of the Room-node. What you gain with this structure is that all children of a node will react on properties given for the parent. So, if the light is off in the Room-node, there won't be any light-calculations, except if you let a light in from a window or door. Another example could be that if you're outside the House-node, it's children won't be visible to you and so they don't have to be drawn. If you then imagine you have several such House-nodes in a city-node you can start to figure out where we're heading. This will greatly simplify the drawing-procedures when you calculate what you can see or not as what you can't see, isn't there, and then it doesn't have to be drawn and when not having to draw a house, we don't need to bother about it's children either. Those things saving quite a lot of memory and cpu-usage. Also when you come to moving things around, most objects are divided into lesser objects, so a car for example would be a car-node with doors, wheels, driver as children-nodes, which moves when the car moves but gives them a possibility to also move on their own, like open the window, turn the wheel. Another thing to save memory is that once you have an object with its textures, materials, meshes and shading you can make new ones as instances saving memory and increase the speed of your game and rendering. To explain this in a simple sense, if you have a knight in a scene, and you want two, you don't quite copy the first one, you just make a instance, two knight-nodes. This means that the two knights share the same basic information such as texture, lightning, material without you actually having to duplicate it. Once you have your tree-structure of things there can also be nodes or node-attributes such as state, position, Level of Detail, or event, making it easier to sort or calculate the viewable field and further speed up the time of rendering and the work for your graphic's card and computer.

2.1.3 What is it used for and who can use it?

The game engines are not only used for, opposed to what the name suggests, games. They can also be used for other applications with real-time graphics such as marketing demos, visualizations of architecture and training simulations. The history of the game engines didn't really pick up speed until the beginning of the 90'thies even though they had existed to some extent even before that. However, around the beginning of the 90'thies the engines started to get recognition and ever so slowly became a more and more usual part in new games. As the popularity to use them grew, big companies created their own game engines and then licensed the source-code, so that when others who liked their ideas and structure wanted to use their engine they had to pay a fee for the use. One example of such an engine is the Unreal (Unreal Tournament) engine on which the game Lineage II is built. There are also a couple of Open Source game engines and Rendering/Graphics engines that any game or software developer can use. The code is available over the net and you're free to add or take away what parts you like or dislike.

2.1.4 What is Crystal Space?

Now that we have gone through the generic parts of a modern day 3D engine it is time to take a look at the actual engine the rest of this study will be focused on, Crystal Space.
Crystal space is a General 3D development kit written in c++, developed using open source and is freely available to anyone. While it have been used mostly to create game applications it is built in such a way that it can be used to create any type of interactive 3d application. It contains not only the renderer used to display the graphics, but it also all the parts crucial to creating a full, working game mentioned in the above section.
The Crystal Space development team has managed to continuously update the engine and add support for new features brought forth by the producers of graphics hardware. Because of this Crystal Space is perhaps the only open source 3d engine able to compete with commercial engines since it offers a complete package (with the exception of network capabilities). One indicator of quality can be the fact that Crystal Space have been chosen as a base for a number of commercial games.

Crystal Space is built from the ground up with modularity and flexibility in mind using a system were the functionality is divided up in smaller parts and placed in "plug-ins". This gives the user the choice of which parts he wants to include in his project and it also enables the engine to be split up into separate pieces so that many developers can work on separate pieces concurrently, as long as the interfaces used between the modules are fixed. Crystal space has also implemented a number of design patterns were applicable in order to utilize established solutions to certain problems instead of designing their own.

Another feature of crystal space is that it supports a large variety of platforms. This fact can greatly reduce the time needed to port a project from one supported platform to another. Currently the list of supported platforms include (among others): Windows, Linux, General UNIX and Mac OS/X.

Now it might seem that Crystal Space only has strong points and no weaknesses but unfortunately that is not the case. One of the drawbacks derived from the modular design is a potential increase in processing overhead which can make the final product operate slower then it might have with an engine written and optimized specifically for that application. This however is often an acceptable trade-off since for many smaller project it is simply not feasible to develop their own engine, either for financial constraints or due to limited development time. Despite not achieving optimal efficiency for a particular application Crystal Space is often quite capable of meeting the performance requirements placed upon it.

Another area that might be problematic is that since it is being developed through open source the task of documentation falls upon the programmers themselves. This often leads to slightly inconsistent and/or outdated documentation. While there is a multitude of tutorials and task specific help available there is little in the way of design documentation for the engine. This is due to that most communication between the designers has been handled through mail correspondence and has never been written down in a structured way. Recently an effort to remedy this situation have begun but so far it has not yet yielded any concrete improvements.

Despite the lacking documentation the support available from the developers can still be described as good. This is due to the quick and accurate response you can get from the developers either through mailing lists or directly via IRC.

3 Design Patterns

So, what's design patterns? A design pattern is a language independent solution to a commonly occurring problem. The underlying meaning here is that when you are implementing huge software or big programs or even smaller, you more or less daily encounter problems that need a solution. It can be small things such as figuring out the best way to structure your program or how to save memory when having multiple objects. Its intent is to create generic solutions to common problems so that the developers do not have to solve the problem again each time they run into it. It also helps the developers explain their program to others, they can simply refer to that they used a specific design pattern to solve the problem and hopefully the other person know that design pattern and how it works. The GoF 'Group of Four', were four people that had recognized those common problems and patterns, or at least realized that there Were common problems in programming and decided to ease the load of other of their kind by using their common knowledge and gather all the problems they could come up with in a book. Those common problems and suggestion of suitable solutions they named design patterns. Design patterns, one could assume, because whenever you implement something you do, consciously or unconsciously follow a pattern, because there Is just a limited way to implement certain things, and thus you follow those swindling patterns into the same kind of problems.

3.1 Design patterns in the Crystal Space 3D engine

The following design patterns are those that we, after some inquiring, were told to be of some use in the Crystal Space engine. Each pattern has a few set points to be answered

- General description
- Case specific problem
- Case specific solution

3.1.1 Factory

General Description

This design pattern can be used when a class do not know the type of object it must create when the program is compiled. It can also be used when you want to delegate the task of creating object to another class and thereby not have to worry about how the actual implementation is handled.

The first step is defining an abstract class defining the interface for creating an object. The type of the object to be returned should not be hard coded into the method call. Then we create subclasses that inherit from the interface class and implement this interface for each specific type of class we want to be able to create. The method called when creating an object is called a factory method because it's the method that actually creates a new object. When an appropriate subclass is called a function of the correct class is returned. If done correctly this decreases coupling between the object that wanted the object and the actual implementation of the same. Other advantages of factory usage include:

- Meaningful names
  In some languages constructors can not have meaningful names which describes what they do, factory methods does not have this restriction.
- Selective creation
  A factory method is not required to create a new object when it is called. Instead it can return an already created object (e.g. when using instancing it can return a pointer to an already created object to save memory).
- Object type masking
  The object returned by a factory method can be of any subtype of the requested type and can require that a client refer to the interface of the returned object instead of its implementation class. This feature enables the API to return an object while masking the classes it uses.
- Forward compatibility
  A factory can return object types that were not yet written when the factory was implemented. As long as all interfaces are followed new classes can be made to use old factories.

Case specific problem

Sometimes you want to have several copies of one object (be they enemies, trees, textures and so on) but at the same time creating the same object several times uses an awful lot of memory. This dilemma is encountered in almost all game engines. Crystal Space is not an exception to this.

Case Specific Solution

In Crystal space the above problem is solved it using the Factory pattern. It is more precisely the selective creation aspect of the factory pattern that is used here. Instead of creating the object each time it is needed it is created the first time and when any subsequent calls comes to create such an object a reference to the first one is returned instead. This technique is often referred to as instancing.

3.1.2 Bridge

General Description

The bridge pattern implements a way to separate what the actual class from what the class can do. This is particularly useful when you need to wait with defining how the class performs the actions until runtime. It also allows you to change one of the two parts without affecting the other. Perhaps a real world example is a good way to illustrate the concept of a bridge:

> A household switch controlling lights, ceiling fans, etc. is an example of the Bridge. The purpose of the switch is to turn a device on or off. The actual switch can be implemented as a pull chain, simple two-position switch, or a variety of dimmer switches. [MD-OM]

Case Specific Problem and Solution

This Pattern has been deemed so useful it has become standard procedure to use the bridge pattern to decouple interfaces from implementation with all modules within Crystal Space. The reason for this is that it ensures that inter module communication is independent from the implementation and only know about the interface. This greatly simplifies the development of plug-ins.

3.1.3 Adapter

General description

The adapter pattern (also known as Wrapper) can best be described as a translator. It takes calls made to one interface and translates them into the format of the receiving interface. This technique is often used to add new functionality to legacy programs without rewriting any of the old code. There are two basic types of the adapter pattern:

- Class Adapter
  The class adapter uses multiple inheritance to facilitate the interface translation. The adapter inherits from both interfaces and therefore knows how to call through both interfaces (or to continue the translation metaphor, it speaks both languages). One of the drawbacks of the class adapter is that if both interfaces have identically named members there will be a name conflict. Also, just because two methods have the same name does not mean they do the same thing so mapping one straight onto the other is not an option. Class adapters can be considered simpler to implement then object adapters since they involve fewer classes.

- Object Adapter
  Instead of inheriting from both classes like in the case of the class adapter the object adapter only inherits the interfaces that the client expects to see and contains a member of the outside class it wraps. In this case, when the adapter translates a method call it simply calls its member of the adaptee class with the appropriately formatted method call. The Object adapter is popular in languages that do not support true multiple inheritances such (such as java). Object adapters have the added advantage of having multiple member classes and thus acting as a translator for more then one object at a time.

Case specific problem

Crystal Space has a lot of support for third party modules. If the developers are to rely on this the interfaces provided by the engine must be kept under the control of the engine development team and not be subject to the whims of a third party.

Case specific solution

In order to accomplish the objective laid out in the section above an adapter is created for each such module in order to provide a constant interface to the users of the engine. This way, if the interface of an outside plug-in is changed all that is needed is for the Crystal Space teem to issue a new version of the adapter with the new function calls instead of having each developer who uses that module rewrite their code.

3.2 Additional patterns

Through mail correspondence with the Crystal Space developers we have been able to discover that the patterns described in this section are used within the Crystal Space project. However, due to comprehensive lack of design documents we have been unable to ascertain their case specific function. Because of this the Patterns described bellow will not contain the case specific problem and case specific solution paragraphs.

3.2.1 Composite

General Description

Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.

Today quite a lot of interactive programs and toolkit software uses this design pattern. Basically you could use it in any software where you can click and choose one or more shapes or objects, group or un-group them and treat them as one entity, independently of their class. A typical use of this pattern is in graphical applications such as drawing editors. To take one such as example, one would have both primitive classes like Text and Line and other classes as well to act as their container, the trick here would be that you want to treat both primitives and containers the same way, at least in some ways. The composite pattern tells you how to create a recursive composition to solve this. The key to the pattern would be the abstract class that represents both a primitive and its container.

This diagram shows a typical composite object structure of recursively composed Graphic objects: Like this picture:

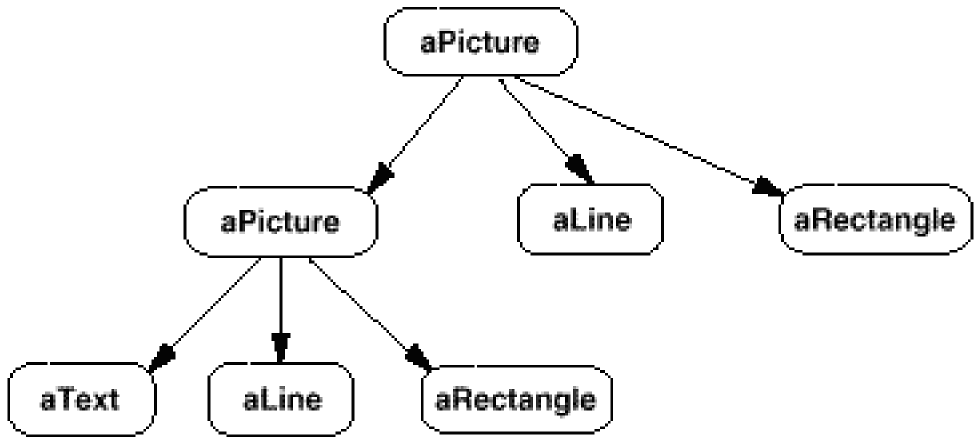

Figure 1: Composite example

aPicture is a container class as you can do more than just draw it. This means it also includes, over it's own draw-operation, operations for it's children accordingly, and as the Picture interface conforms to the Graphic interface, Picture objects can compose other Pictures recursively

Clients use the Component class interface to interact with objects in the composite structure. If the recipient is a Leaf, then the request is handled directly. If the recipient is a Composite, then it usually forwards requests to its child components, possibly performing additional operations before and/or after forwarding.

Even though the creators of Crystal Space vaguely mentioned it is used in a number of places, one of the few they could point at were the Window Handling-functions.

3.2.2 Iterator

General description

Also known as: Cursor. Definition: Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.

An example of such an aggregate object could be a list, which is a quite easy structure and very usable in many areas. Operations what to do with this list and how to go through it might depend a bit what you actually want to do with the list in the first place but even so you might want to have operations for all eventualities in your software. This pattern makes you do this, but instead of putting all possible operations and traverse-methods into the actual list-object this pattern will put it into an iterator object and so take responsibility for the actions of taking out, in new objects in the list and traversing it. In the iterator-class it will then create an interface for access to the list-object and the iterator-object is responsible to keep track of he current element, basically keeping track of what parts have been traversed already.

The basic point of actually separating the list-functions from the actual list-class is to make it easier to use different traversal methods without having to write them all in every list-class. Also, with such a structure you can make something called polymorphic Iterators, allowing you to use the same methods for different kinds of lists, making it possible for a client to traverse any list in the same way.

3.2.3 Observer

General Description

The observer pattern (also known as: Dependants, Public-subscribe) is used when many object need to be notified whenever the state of a single object is changed. It is a definite one-to-many dependency. This solution is realized through subscriptions, each of the receiver objects calls a method on the sender to notify the sender of its existence. When a change occurs in the object to be monitored the sender delivers this information to each object registered in its list of subscribers.An often used example of this is when you want to separate the data from how it is displayed. An example in order to illustrate its use might be a spread-sheet program. Here you can have a multitude of different diagram options in addition to the traditional spread-sheet view. In order to avoid misinformation in the diagrams they must be updated with new data whenever the spread-sheet data is changed.

4 Benefits and Drawbacks

The benefit of using design patterns is that you avoid making common errors when you code and that you can structure your code in a much better way and also, use the code more efficiently. With those design patterns you can get some good examples on how to solve problems and how to save precious memory and cpu-usage. One drawback of the design patterns can of course be that you get stuck in a already structured and clearly pointed out path. This will of course prevent you from making huge errors and waste days of going through badly structured code, but it could also somewhat damage the creation of other possible solutions to those common problems. Also, using the design patterns might in some areas give you more work than what finding another solution might do, however as far as we can tell those occasions are sparse and easily accounted for.

5 Conclusions

Further developments

While this study tries to cover how design patterns are used within the Crystal Space engine the limited time available of writing this report prevented us from studying the engine as closely as we would have liked. The engine has been in development for over 6 years and the program has grown at such a rate that any effort to write design documentation have failed. At this stage the detailed design of the engine can only be found through dialog with the developers together with in depth code study.

Final word

The conclusion we were able to draw from our case-specific study would be that yes, as a whole the developers of the Crystal Space engine have gained some advantages from their use of design-patterns but there probably still are places in the code were a design-pattern solution would have been beneficial. The reason they have not been found yet is due to the extensive lack of documented design. If a compressive design description were to be made, additional optimizations would probably be detected. there probably will would probably have gained more with more extended use of design patterns as well as they would have gained quite some on having well-made documentation for the design and some sort of history concerning the forum and mail-communication they have had.

6 References

References

[GOF95] E Gamma, R Helm, R Johnson, J Vlissides *Design Patterns: Elements of Reusable Object-Oriented Software*, ADDISON-WESLEY, 1995

[DOF] Data and Object Factory website *http://www.dofactory.com/Patterns/Patterns.aspx*

[GEA101] Game Engine anatomy 101 *http://www.extremetech.com/article2/0,1697,594,00.asp*

[WIKI] Wikipedia: Game engines *http://en.wikipedia.org/wiki/Game_ engine*

[SG] Scene graphs: Avi Bar-Zeev *http://www.realityprime.com/scenegraph.php*

[MD-OM] Michael Duell, *"Non-software examples of software design patterns", Object Magazine, Jul 97, p54*