ASD Assignment: 3D Game

December 10, 2012

Carlos Pérez López

Contents

1 Initial Research

Among all the options for the ASD assignment, the simple 3D video game was chosen. The author found all of them of real interest, but two reasons led him to take that decision. The first one is his unfamiliarity with the CGI field; even though he counts with a quite solid basis in computer engineering, all the concepts such as shader, raytracer, renderer, flocking system, etc. are new for him. Thus, instead of a more specific project, it was thought that at the same time lectures give a more theorical vision of computer graphics, a simple 3D game might be a good choice to get familiar with all of it and adquire a more global vision. And independently of the academic aims, the second reason was a quite well-formed idea of a 3D game in the author's mind.

By the time of starting the research, different videos of some 3D games were watched as well as reading the documentation of some libraries commonly used in the field of video game programming. Although Qt offers a very complete API, is more focused to the development of applications which need a powerful and friendly GUI. On the other hand, SDL offers funtionalities more suitable referring to game programming. The SDL library [SDL 2.0, 2012] is completely compatible with 3D harware acceleration via OpenGL, and provides a series of strong subsystems such as input events, threads, timers, audio, etc. Other of the main advantages of this library are that it supports the use of joysticks, which is really convenient in the world of videogames, it is cross-platform and completely open source.

Some other libraries were studied for their use as well. BulletPhysics [BulletPhysics, 2012] was thought in first instance as main physics engine to handle the collisions in the world of the game due to it is a full package which offers a wide range of collisions detection. In the same way, to implement the AI of the system, the author had in mind OpenSteer [OpenSteer, 2012], which offers steering behaviours for autonomous characters in game and animation. Finally, due to the not very high complexity level of this project both libraries were rejected. Besides, using directly low level C++ to implement it, the approach of the used algorithms would be greater, so it would be a better chance to learn.

The final technologies chosen to implement the game were NGL, the graphic library of the NCCA, and SDL2 for handling the graphic context and the user events. Later, SDL2_ttf would also be added for debugging purpose and because some functionalities of rendering text are required at some points of the game play.

The main source of learning NGL was the ASD professor website [J. Macey's website, 2012] as well as his blog [J. Macey's blog, 2012]. Reading the demos' code and writing small programs was the process of understanding the concepts a progressing little by little. The same happened

with SDL, but extending it with all the other sources that might be found through the internet.

Parallel to this, other research was being carried out relative to the design of the game which, certainly, is completely independent of the technologies chosen to implement it. M. Sanchez provided to the author the contact of E. Anderson, an experienced teacher in games programming working at BU at the moment. He gently gave to the author an explanation of the typical game architecture, as well as provided some slides he often uses during his lectures.

At this point, the research material and documentation was enough to start with the design of the 3D game.

2 Initial Design

Currently, the most part of applications are desgined using the paradigm of Object Orientation, due to the flexibility and scalability that design patterns offers, as well as it is widely belief as a very smart way of programming. But concerning to this project, it must be added that the execution of a game is normally a very secuencial flow.

All the games has a main loop with a similar estructure:

- reading the player input
- updating the world
- checking the collisions
- drawing the world
- adjusting FPS

So to design this execution flow, it was decided to count with a class *GameManager* which will be the centre of the system and will mantain the specific behaviour of this specific game. The rest of the modules which are surrounding this master class will purely have an object oriented design and will conform our game engine.

A game engine is a framework designed for the creation and development of video games. It might be thought as a group of predefined functions that offer to the game developper behaviour already programmed. Thus, all the different modules in this game should be part of a game engine, the most independent possible of this specific game, and that might be used in the future by other games of similar characteristics.

Once these concepts were clarified and the main structure was decided, the different modules had to be thought. One the main worries of the author at the time of the design were

the flexibility and the scalability, so a convenient idea was treat every module as an independent system which is charge of certain responsabilities logically grouped and that could unacoplated and used by any other client. See 1.

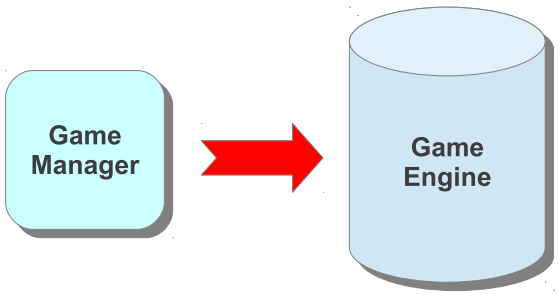

Figure 1: Game engine scheme

In order to define the world, a class *Object* was defined, which represents a simple object in the world and that mainly stores state information and some little behaviour for the interaction with other objects. This is the top of the hierarchy, and then some other classes with more specific state and behaviour might be defined. To store the objects of a concrete world there is manager named *ObjectManager* which holds a list with all the objects a provides an interface with basic operations such as *addObject* or other more linked to this context.

The module in charge of updating the world is born in the class *Controller*, which provides a hierarchy of different autonomous behaviours (the AI of the game) as well as the controller defined by the user input. At first, this was thought as a strategy pattern, since the objects might have different strategies of movement but later, regarding the fact that is the controller the entity that moves an object and will have to modify its state, it was decided to use composition. So, in the same way, we count with a manager class that deals with all the controllers in order to give the client abstraction and to avoid they manage each one of them manually. The figure 2 shows this.

One of the critical points of computer games is the check of collisions among the objects moving around the world. In this approach, it was decided that the best option is that the responsability of checking the collisions should be carried out by the holder of the objects but, using a phyics engine object provided by the client. The *ObjectManager* just deals with a *PhysicsEngine* object which in this system is an abstract class and might be thought as an interface whose implementation depends on the specific physics engine used. In this way, this design provides a very high flexibility. See 3.

Concerning to the graphic module of this assignment, which is one of great importance

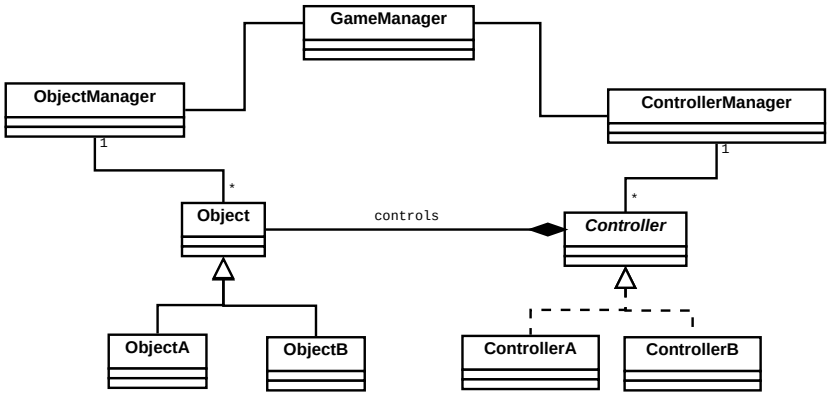

Figure 2: Controller-Object composition

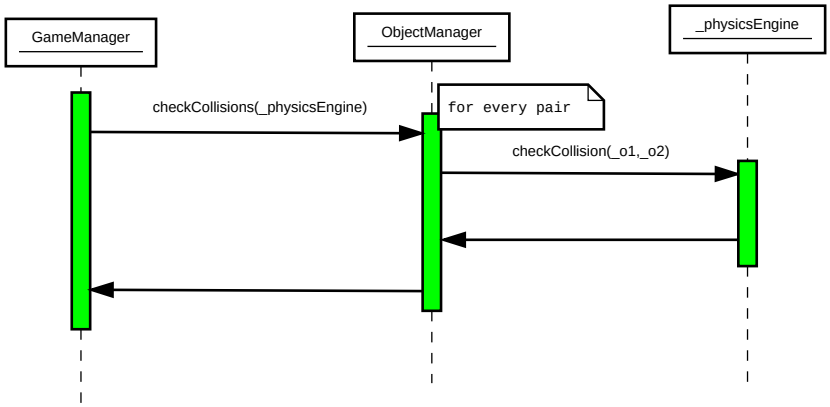

Figure 3: Secuence of the collisions check

taking into account the nature of this course itself, there is class that is in charge of all the funtionalities related with the drawing, which could not be named in other way that *Renderer*. The *Renderer* is the responsible of creating the graphic context and initialize everything needed for the rendering of the current scene of the game's world. It deals with the shaders and lights, and it is thought to be able to draw any 3D object over a background as well as images 2D or text. At the time the client asks for the render of the world, they should provide a reference to the world and its background, the camera from where the render should be processed and a simple flag to indicate a *debugMode*. As a design decision, the cameras are treated apart and independently to offer the user more transparency and versatility to visualize the scene. A *CameraManager* is added to the design as well to update all the positions of the different type of cameras according with a target, that is the main object of the world, the one the player is

controlling.

The design counts with a class which acts as a stock and stores all the sources needed for the excecution of our game, such as meshes, textures, images, text, sound, etc. This is the *SourceManager*. The presence of managers is quite abundant in this system, the reason is that there will be a big amount of different objects during the runtime, and they can offer abstraction to the *GameManager* to deal with all of them. At first it was thought to make them singletons, but finally it was taken the option of just keeping one single instance of each manager.

Finally, in order to load our game world, a *Parser* is used to read from a describing scene file and some auxiliar functions used by the *GameManager* were grouped in a *Utilities* class

3 Proof of Context

4 Future work

References

[SDL 2.0, 2012] Lantinga, S (2012). Available on: `http://wiki.libsdl.org/moin.cgi` Accessed December 10, 2012.

[BulletPhysics, 2012] Available on: `http://bulletphysics.org/wordpress` Accessed December 10, 2012.

[OpenSteer, 2012] Available on: `http://opensteer.sourceforge.net` Accessed December 10, 2012.

[J. Macey's website, 2012] Macey, J (2012) Available on: `http://nccastaff.bournemouth.ac.uk/jmacey` Accessed December 10, 2012.

[J. Macey's blog, 2012] Macey, J (2012) Available on: `http://jonmacey.blogspot.co.uk` Accessed December 10, 2012.