COMP 3501: Team Project Report
Asteroids Game

Due on December 5, 2011

Drew Martin 100784541
Nicolas Porter 100760059

Contents

List of Figures

List of Tables

1 Introduction

1.1 Game Objectives

Our game is a minimalistic 3D space shooter. The player controls a spaceship in the vicinity of the Earthand must blow up asteroids as they move through space by shooting them with a laser. When hit, asteroids split into progressively smaller chunks. The player can adjust their orientation and velocity to get closer to an asteroid and make hitting them easier. Asteroids move with a fixed velocity.

1.2 Game Unique Items

The game contains a robust and flexible component system that allow run time changes to a game object's graphical representation of an object, as well as how it reacts to input and how physics are applied to it. It also simplifies development by making different classes interact with each other in a set way. For instance to perform collision detection, two physics components run tests on each other. It also increases the testability of our game as we can use components specifically crafted for testing.

We also have a unique camera that follows a particular game object, basing its orientation and position on those of the object. It can display the object in either first or third person view. When in third person view, it uses an offset to display partway behind and above the object. It also keeps a queue of the last 10 rotation quaternions the followed object used, and then bases its orientation on the head of the queue. This effectively allows us to display the ship's rotation in the game world, even if there are no objects in view from which the player can make a reference, which is a common occurrence in a game set in space. This effect only happens in third-person mode.

Additionally, our game features a Skybox. A Skybox is a background image which gives the player the illusion of a huge 3D world. This lets us simulate the vastness of space and have a nice looking backdrop, full of stars and nebulae.

1.3 Document Organization

Section 1
Provides a brief overview of the project and discusses some of the main features of the game.

Section 2
Goes into a greater description of the features and gameplay.

Section 3
Discusses the features included in the game, and those that were devised but not included.

Section 4
Will discuss the software architecture of the game.

Section 5
Discusses the team work behind the project, including the time spent on it.

Section 6
Features our self-assessment of our grade and our justification.

Section 7
Will summarize our work.

2 Project/Game Overview

2.1 Playing the Game

As per the scope of the project, the game is very simple. It has very little gameplay, and serves more as a tech demo to show the functionality we can create. In the game, the player controls a spaceship using the keyboard. To control the ship, the player primarily uses the arrow keys and the "wasd" keys, as well as a few others summarized in the table below.

Key	**Action**
Left arrow	Turns the ship left.
Right arrow	Turns the ship right.
Up arrow	Pitches the nose of the ship downwards.
Down arrow	Pitches the nose of the ship upwards.
W	Increases the ship speed.
A	Rolls the ship counter-clockwise.
S	Decreases the ship speed.
D	Rolls the ship clockwise.
<	Cycles backwards through asteroids to be followed by the camera.
>	Cycles forwards through asteroids to be followed by the camera.
/	Makes the camera follow the ship
1	Sets the camera to first-person mode
3	Sets the camera to third-person mode
Space bar	Shoot a laser
Escape	Closes the game

Table 1: Key bindings

The player can fly around and shoot asteroids that drift through space. When a laser collides with an asteroid, the asteroid splits in two if it sufficiently large. If not, it simply disappears. There is no limit on how much the player can shoot, and no bounds on where they can fly. In addition, the player cannot be killed and will simply fly through asteroids.

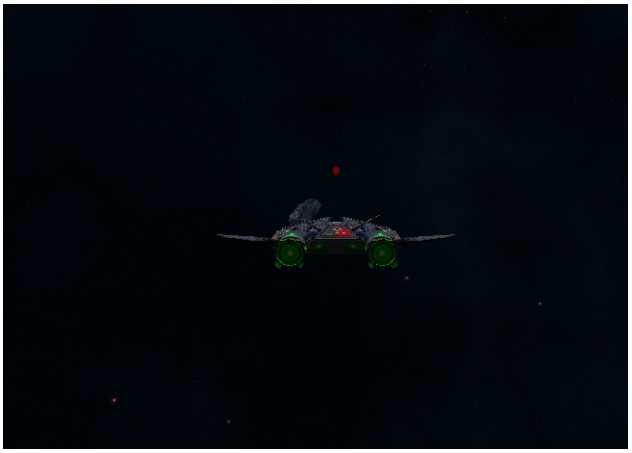

Figure 1: The ship shooting a laser

2.2 Build Requirements and Instructions

This game requires Microsoft Windows 7, Microsoft Visual Studio 2010 or greater, as well as the Direct X SDK. Our submission contains a project file which should be opened directly from Visual Studio. The project does not require any additional libraries. There are no special steps involved in compilation, assuming that the path to the Direct X SDK headers and libraries are properly configured. No special hardware is required to run this project.

3 Special Features in the Project

3.1 Major Features

3.1.1 Dynamic Follow Camera

Our game incorporates a camera that does not store its own position or orientation, but instead bases it off of an object that it is following. We have a MoveableGameObject class, and the camera can follow any object of this class. The MoveableGameObject class and all of its subclasses have the getPosition() and getRotationQuat() methods which return a position vector and a quaternion, respectively. The camera uses those to create the view matrix. The object that is being followed by the camera can be switched on the fly. Just to demonstrate this, using the < and > keys will cycle through the asteroids, causing the camera to follow them. Hitting / causes the camera to follow the spaceship.

The camera can show objects in first-person or third-person view. In game, first-person view can be enabled by pressing 1, and third-person view by pressing 3. The camera has an offset vector which is used in third-person mode to move the camera a fixed distance away from the object prior to performing any other world transformations. For example, if the camera offset was $(0, 1, -3)$ then no matter where the followed object was the camera would be behind it and slightly above, oriented to look just ahead of the object. When the camera is in first-person mode it simply ignores the offset and uses the same position as the followed object.

As the game is set in space, and there are not many objects with which the player can orient themselves and determine in what direction the ship is moving, we wanted to make it easier to see that the ship was rotating. To do this, we made the camera keep a queue of the last 10 rotation quaternions used by the followed object. Then, when transforming the cameras coordinates, instead of using the current rotation quaternion of the followed object it uses one from 10 frames prior. In this way, it is readily apparent that the ship is rotating. This is only done in third-person view.

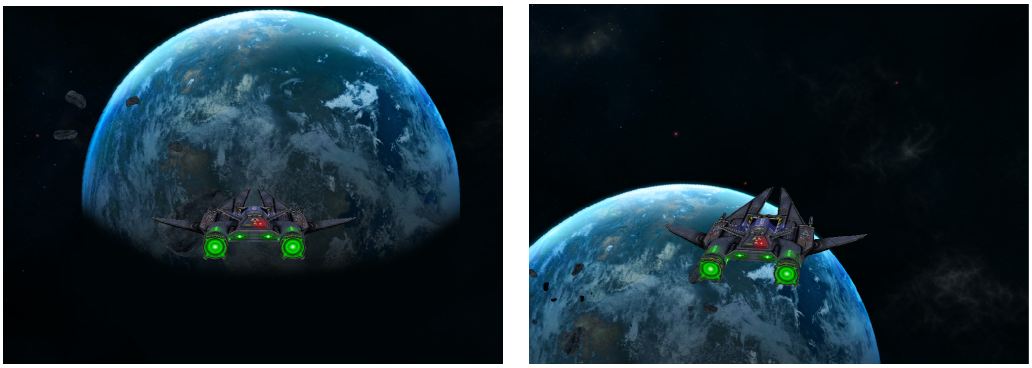

Figure 2: The followed object without rotation (left) and while changing pitch and yaw (right)

Similar to the rotation effect, we also increase the field of view as the followed objects speed increases to make it more apparent to the player that the ship is moving fast (or slow). However, unlike the rotation queue, this takes effect in first and third-person views. The angle varies between 45 degrees (when stationary) and 70 degrees (when at the maximum speed for the object).

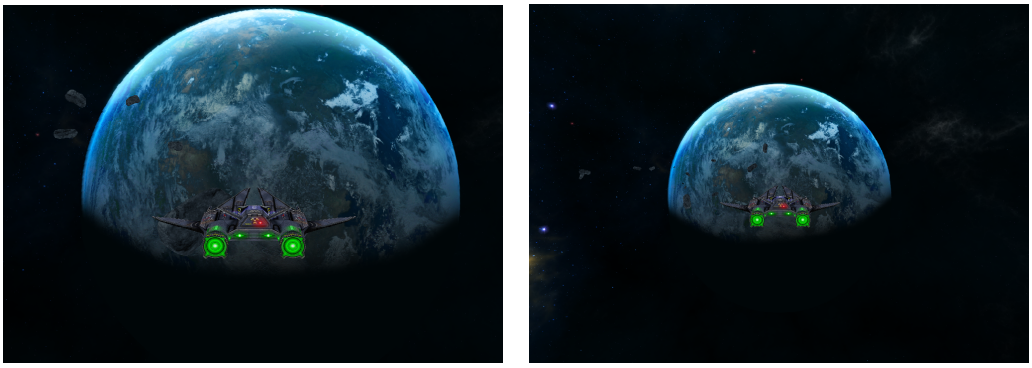

Figure 3: The spaceship while stationary (left) and at maximum speed (right) showing the difference in the field of view

3.1.2 Skybox

A Skybox is a background image that adds realism to a 3D scene by giving the player the illusion of being in a huge world. This effect is achieved by projecting an image on the inside of a cube and centering that cube on the camera's position. This means that the player can never reach the skybox's edges, giving the impression that the background is far away.[1] Although, some more advanced skyboxes contain models, animation and lighting, our limited time schedule allowed us to implement a skybox that only supports textured quads.

A skybox can be implemented by simply building a cube with all of its faces pointing inwards. This causes the textures to be drawn on the inside faces instead of on the outside. Before rendering the skybox, you must disable the Z-buffer, since we want it to be the farthest thing rendered. Also, since the skybox is supposed to be far away from the world, you do not want it to be affected by the same light as your models. Therefore, lighting should be disabled when the skybox is rendered.

Since the player should not be able to reach the skybox, the skybox is centered on the camera's position.

Alternatively, the skybox can be snapped to the far clipping plane. This is achieved by setting the w component of a vertex to zero, then multiplying the vertex with the world matrix and the projection matrix, as usual, and finally setting the results w component as its v component. After the perspective divide, the z component of the vertex will be equal to one, which corresponds to the far clipping plane.[2]

3.1.3 Laser-asteroid Collisions

Lasers are shot from the ship by pressing the space bar. If a laser collides with an asteroid, it will cause that asteroid to be destroyed. If the asteroid was large, then it will also create two asteroids in its place. These smaller asteroids start from the same position, and a random direction will be generated for them. One asteroid will travel in this direction, the other will travel in the negative of that direction.

Collisions between lasers and asteroids use a brute force mechanism. Every frame, all asteroids are checked against all lasers to determine if they collide. The collision detection does very simple calculations to determine if there is the possibility of a collision, and progressively does more complex, time consuming calculations until it ultimately determines whether or not the laser and asteroid collide.

Lasers keep track of their current and previous positions, and from this a line can be drawn that covers the exact space that the laser travelled in the last frame. A ray can also be created from the lasers position last frame along its current direction (which we will call last ray), and another ray using the current position and direction (which we will call current ray). The collision detection algorithm first checks if the ray intersects the smallest sphere that completely contains the asteroid. If not, then the laser has not collided with the asteroid and we are done. If the ray and sphere do collide, it means we may have a collision between the two. The sphere-ray collision check is done with the D3DXSphereBoundProbe() function.

Next, we want to find if the asteroid falls on the line between the lasers last position and the lasers current position. There are three situations for this (assuming the last ray intersects the sphere), as seen in Figure 4:

1. The line intersects the sphere in two spots, meaning the line passes through the asteroid. In this case, the last ray will intersect with the sphere, but the current ray will not.

2. The line ends inside the sphere. In this case, the last ray intersects the sphere and the current ray may or may not detect this as a collision. However, the distance from the lasers last position to the asteroid will be less than the radius of the sphere in this situation.

3. The line does not intersect the sphere. In this case, both the current ray and last ray will intersect the sphere, but the distance from the current point to the asteroid is greater than the radius of the asteroid. This indicates that there is no collision.

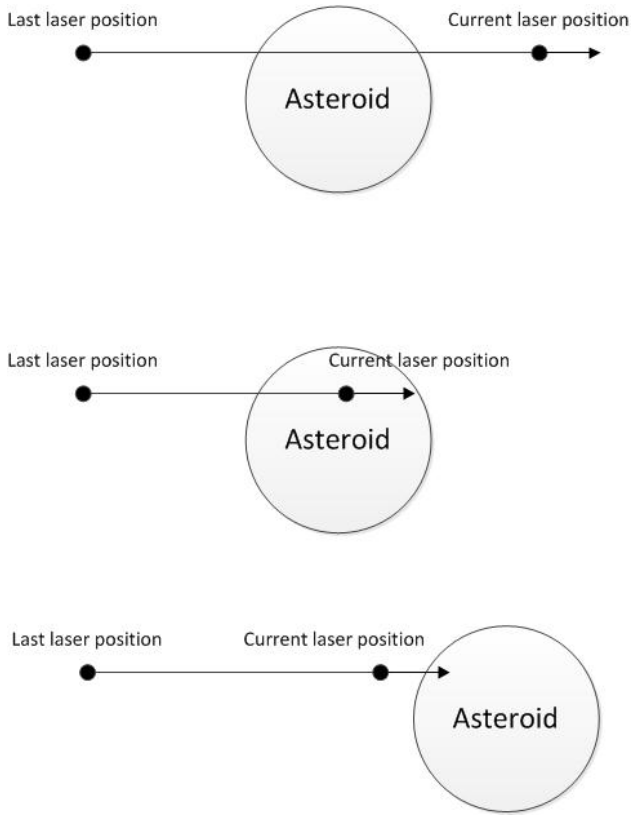

Figure 4: The three possible collision scenarios
From top to bottom:
The ray from the last laser position intersects the sphere, but not the ray from the current position.
The ray from the last position intersects the sphere, and the current laser position falls within the sphere.
Both rays intersect the sphere.

If the third case happens, there is no collision and we are done. If either of the first two happened, we continue checking. Next, we check whether the last ray collided with the asteroid mesh. In addition to telling us whether or not they collided, this will also tell us how far away from the ray's point of origin they first collided, if they do intersect. If the ray and the mesh intersected, and the distance from the last laser position to the point of collision is less than the distance from the last laser position to the laser position then we know that there was a collision and we handle it as such. If either of these are false then there was no collision this frame and we are done. The ray-mesh collision check is done with the D3DXIntersect() function.

3.2 Minor Features

In addition to the above we also had some minor features that are still notable.
The sun is drawn like the skybox, by first disabling the Z-Buffer and drawing it relative to the player position. In this way, it is unreachable by the player and appears to be very far.

The Earth is drawn in the same way as the sun, except that it's not always the same distance from the player. It is drawn near the player, and as the player travels towards the Earth it is rendered a certain distance from them. However, this distance is not constant. As the players travels towards the Earth, the distance slowly decreases such that they can reach it if they fly far enough. Doing this allowed us to created the illusion

that the Earth is much larger than it actually is in the game world, but still keep it as part of the background.

We used lighting in the game as well. There is a moderate-strength ambient light, as well as several point lights and directional lights. There are several direction lights coming from roughly the direction of the sun to simulate it bathing space in light. There is also a strong point light whose point of origin is roughly where the sun appears to be in the world. There is a weaker point light at the Earth's position, representing the large quantity of sunlight reflecting off the Earth into space.

3.3 What was not accomplished

We had brainstormed more features than we planned on actually putting into the game. If we had extra time, or other features took less time than anticipated, we would have added more. Some of the features we had envisioned, but ultimately did not implement, were:

1. Rendering the Earth as a sphere, potentially with bump mapping, light mapping and reflection mapping.
2. Collisions between your ship and an asteroid, destroying the ship.
3. Collisions between asteroids and Earth, causing damage to the Earth.
4. Changing the appearance of the Earth based on how much it had been hit by asteroids.
5. A nicer looking laser, using shaders.
6. Particle effects when the laser contacts an asteroid.
7. Particle effects from the engines of the ship.
8. A crosshair to display where the lasers will hit.

3.4 What was hard

The collision detection was somewhat difficult to do in a way that was not too inefficient. Even as it is, we have to limit the number of asteroids in the game at any time to prevent the game from running too slowly. If we had more time, we'd try to optimize the collision system and physics components such that the collision detection would not use a brute force approach.

The camera itself was not terribly hard once the concept of quaternions was understood. It was somewhat time consuming to make because the game objects had to be refactored to use quaternions to represent their rotation, instead of just using their direction and up vectors.

Designing an architecture which does not contain any singleton proved to be harder than expected. Our solution was to pass the game engine and the graphics engine as parameters instead of introducing global mutable state for the sake of simplicity. This seemed messy and annoying on paper, but after implementing it, this solution felt natural and made our code more predictable.

It was hard to figure out the proper way to use method pointers, for the input system. We had to rely on casting to make it work. The type safety is ensured by the KeyboardInputHandler interface. We also tried to implement this feature using C++11's function objects, but we could not get it to work properly.

4 Game Design Layout

4.1 Software Design

4.1.1 Game Objects

A game object is used to store the state an entity within the game world. It also contains a repository of game components which modify its state.

Game Object	Description
MoveableGameObject	Superclass of all game objects that can move. It stores such things as speed, direction and rotation.
Asteroid	Represents one asteroid. Inherits from MoveableGameObject.
Laser	Represents one laser. Inherits from MoveableGameObject.
Laser	Represents the spaceship. Inherits from MoveableGameObject.

Table 2: Game Objects

4.1.2 Game Components

The game objects that make up our game world are composed from game components. This system avoids the tight coupling of graphics, physics and input code found in classical game object hierarchies. A component-based approach to game objects gives you better modularity and maintainability.

Indeed, all that must be done to change an object's graphical representation is to swap its graphics component. Furthermore, if we ported our project to OpenGL, we would simply need to modify the graphics components of objects (as well as the graphics engine, of course) since nothing else depends on Direct X. Also, if you want to allow the player to control an object, you simply give it some input component that deals with keyboard input. If you want to test an object and require that object to move deterministically, then you can give it an input component which simulates key presses. Finally, these components can also set runtime. This system is very flexible, as shown by figure 5.

Figure 5: The graphics component of our asteroids has been replaced by the teapot graphics component.

Our implementation supports three types of components: Graphics, Input and Physics. It is based on the implementation found at `http://gameprogrammingpatterns.com/component.html`.

Graphics Components
Graphics components store the graphical representation of an entity and draw them.

Component	Description
AsteroidGraphicsComponent	The graphics component that loads, stores and draws an asteroids mesh and texture.
LaserGraphicsComponent	The graphics component that stores and draws a laser mesh, which is generated by the D3DXCreateCylinder() function.
SpaceshipGraphicsComponent	The graphics component that loads, stores and draws a spaceship's mesh and texture.

Table 3: Graphics Components

Input Components
Input components allow entities to react to keyboard input. This class implements the keyboard input handler interface which allows callback registration with the input system.

Component	Description
PlaneInputComponent	Maps controls to ship movements.

Table 4: Input Components

Physics Components
Physics components are used to move the objects every frame and detect collisions.

Component	Description
MoveableObjectPhysicsComponent	Controls the movement of all MoveableGameObjects
LaserPhysicsComponent	Handles the collision detection of lasers and asteroids

Table 5: Physics Components

4.1.3 Game Engine

The GameEngine class drives the game. It contains the game loop, as well as references to the game app and graphics engine.

4.1.4 Graphics Engine

The GraphicsEngine class provides access to the Direct3dDevice and provides wrappers to some commonly used functions, like mesh loading and setting the background color. It also provides a way to add scenery elements, and renders them. If we would have had more time, this class would have provided a full abstraction over Direct X.

4.1.5 Scenery Elements

Scenery elements are elements that are not reachable by the player. They are the first objects to be rendered to the scene. Both the Sun and the Skybox are scenery elements.

4.1.6 Input System

The input system allows classes to register callbacks for key presses. We do not use Direct Input because it has been deprecated in favor of the Win32 API message loop. However, Win32 API's input support is very primitive compared to Direct Input. Furthermore, it does not specify a function equivalent to Direct Input's GetDeviceState, thus this functionality was implemented in the input system.

As figure 6 shows, the input system registers callbacks with the Window System for the `WM_KEYUP` and `WM_KEYDOWN` messages. The key that was either pressed or released is stored in the wparam argument of the message. The input system then sets the value of the key to true or false in a buffer (KeyboardBuffer), depending on if it received a `WM_KEYDOWN` or `WM_KEYUP` message, respectively. During the update loop, the game engine tells the input system to call all callbacks registered to currently pressed keys.

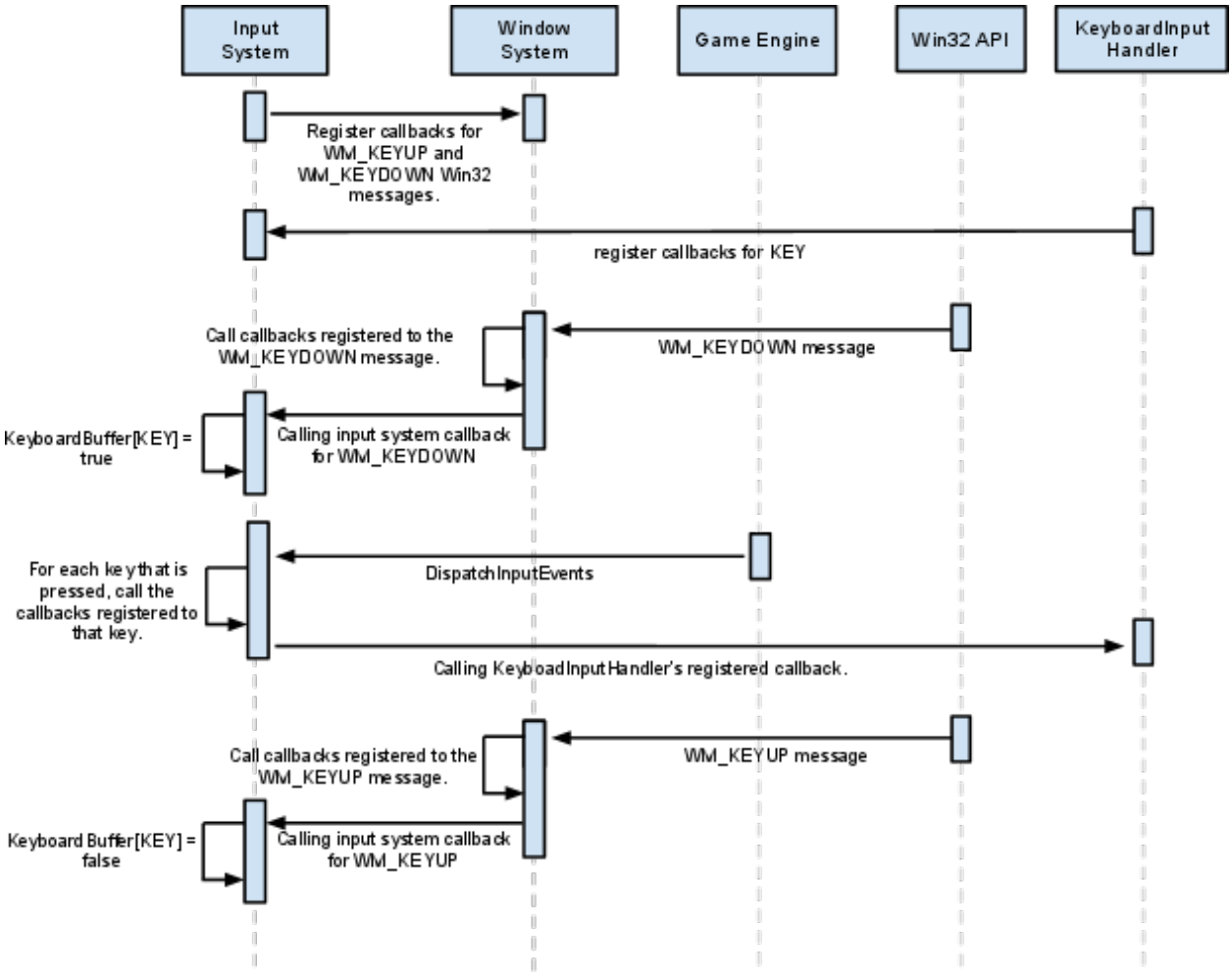

Figure 6: Input System sequence diagram

Only classes that implement the Keyboard Input Handler interface are allowed to register callbacks with the input system. Registration is done at runtime. Furthermore, our input system implementation is limited to keyboard input, as we did not require mouse support. However, it could easily extended to work with different input methods.

4.1.7 Game App

A game app is an abstract class that specifies the low level details of the game, namely its title and window size. Its purpose is to hide the low level initialization routines such that derived classes only need to implement three methods: init, update and draw. The game app also implements the Keyboard Input Handler interface in order to close the game when the escape key is pressed. Classes which derive from the GameApp class should be in charge of handling game worlds, and managing the game objects that should persist between game world instances. This has a nice performance benefit: we could initialize the game's graphics components in the game app, and pass them to the game worlds. In essence, it allows you to store all the state that should be shared between game worlds.

The AsteroidsGameApp class, a derived class of GameApp, contains a reference to the game world and is responsible for calling update on it, which in turn is called by the game engine. Additionally, it handles certain input that is not tied to a specific game object, such as changing which object the camera is following.

4.1.8 Game World

Any class who implements the game world interface should be considered a level, or world of the game. Since our game does not need levels, we only have one game world, the AsteroidsGameWorld class.

4.1.9 Window System

This class is an abstraction of a Win32 window. It allows other classes to register callbacks for Win32 messages. Its callback registration mechanism is very similar to the one found in the input system.

4.2 Game Human Interaction

There is very little in the way of UI in our game. The top left corner contains position information which is more for debugging than anything. We would have liked to have a crosshair to help the player with aiming but did not get around to it.

We chose to have the player control the game with the keyboard as it was simple. At one point we played with the idea of having keyboard and mouse control, with the mouse affecting the yaw and pitch of the ship, as well as clicks controlling the shooting. However, we ultimately decided on keyboard control alone as mouse control might feel strange for a game set in space. The more traditional control would required a joystick, however since neither of us have one this not considered.

5 Team Work

5.1 Team Approach to the Project

The work was split among the team based on the strengths of the members. Nick, who is more artistically inclined, took care of creating the backdrops for the game world, as well as implementing the game engine. Drew, who is more mathematically inclined, did the physics of the game, including movement and collisions.

We did not run into any issues as there were well-defined goals from the start and the work was split up well.

Task	**Estimate**	**Actual**	**Reason**
Overall Effort	47h	70h	
Design of game	2h	2h	We intentionally kept the design simple, as per project instructions
Design of code	47h	70h	Mostly designing the component system. Other features were basically made ad-hoc within the confines of the component system
Implementation	40h	63h	Went overboard with the abstractions (Graphics Engine, Window System). We added more features than originally planned (more complex camera, skybox)
Integration and testing	0h	0h	We tested as we added features, not in a separate testing phase. Integration was a matter of performing merges with git.
Follow Camera	8h	8h	This required refactoring MoveableGameObject and everything that implements that class.
Component System	10h	12h	Was fairly straightforward but it was hard to hook it up in a way that didn't require global variables or singletons for the game engine or graphics engine.
Input System	5h	15h	Had to deal with the shortcomings of the Win32 API. Also had problems with method pointers as callbacks.

Table 6: Time estimates

5.2 Integration

No major integration problems were experienced during the development of this project. Apart from our Game World class, we were mostly working on different files. We used a tool called git for version control and we made sure to fetch and merge often. A combination of testing new features as we added them and the robustness of the component system meant that the integration was painless and we ran into no issues.

6 Grade

We believe our mark for the game is 100% due to the component system, dynamic follow camera, skybox and physics. Had we not believed we deserved 100%, we would have done more work.

7 Conclusion

In our asteroids game we managed to implement several things related to the course objectives: lighting, 3D camera and movement and meshes. We also did some things not directly related to the course but still very useful for game development, notably the component system and physics. Use of the component system allowed us to build on a well-architected system which made development easier and helped reduce potential bugs.

If we were to continue the development of the game, we would definitely try to optimize how the physics components interact with each other to render collision detection more efficient. We would add asteroid-to-asteroid collisions and asteroid-to-ship collisions. Also, we would add particle effects when a laser collides with an asteroid, and particle effects coming from the ship's engine. We would also make the Earth a sphere instead of drawn on a quad.

If we were to develop another game we would definitely spend as much, if not more, time on a well architected system as that has helped immensely. Instead of trying to write physics from scratch, we would use a library to help handle it, as its implementation would likely be more efficient than what we have managed to. We would also include the Boost library, as it has many features which would enhance the component system such as its boost::any data type, a variable of any type.

We were happy with what our game allowed the player to do, to fly around space and blow up asteroids. Considering it was supposed to be a small project, we are happy with what was accomplished.

References

[1] Build A Skybox. http://en.wikibooks.org/wiki/Blender_3D:_Noob_to_Pro/Build_a_skybox. Retrieved Monday December 5th.

[2] Koen Samyn. 3D Skybox. http://knol.google.com/k/3d-skybox, 2009. Retrieved Monday December 5th.