DirectX Scene Technology Demonstration

Coursework Report for AG1101A
University of Abertay, Dundee

Sarah Herzog
18 Jan 2013

Introduction

Computer graphics have grown in leaps and bounds over the history of game development, and with graphical improvements comes new and improved graphics libraries. While games programmers can often rely on graphics APIs which simplify much of the graphics programming process, understanding the graphics pipeline is vital in order to take full advantage of the display firepower of modern computers. This report and project focuses on the demonstration of many features inherent in the DirectX 11 graphics library, with a particular focus on it's application in field of game development.

The demonstration project is presented as a space game in which the player pilots a shuttle exploring the Sol system. The program exhibits many basic DirectX game-related features. These include the following:

- Loading and rendering models from file
- Transforming the world, view, and projection matrices to render models in different locations or rotations
- Ambient, diffuse, and specular lighting
- Multiple textures
- Camera control using the keyboard
- Frames per second and CPU usage counters.

In addition to these, several advanced features have also been implemented. These include:

- 2D texture rendering for title screen, control screen, HUD, and mouse cursor
- In-scene point lighting (as opposed to single directional lighting)
- Sky-box starfield background
- Particle system comet tail
- Direct Sound music and effects

The rest of the report will examine these features and their implementations in detail, discuss the structure of the program and code, appraise the effectiveness of the program and design decisions, and reflect on lessons learned through the project's completion.

Features

The demonstration application takes the form of a space exploration game. The game, dubbed "Sarah's Smashing Space Sojourn", puts the player in control of a shuttle exploring the Sol system. Attention was paid to representing planets somewhat realistically, with the idea that the game could be used in an educational application in the future. While planet orbit radii were reduced drastically to keep the scene easily explorable, all other dimensions were kept in proportion. This include body radii, axial tilt, rotation speed, orbit speed, and orbital inclination. The only body whose radius was not preserved was Sol itself - the radius was reduced in order to make viewing the planets easier. Planetary details such as radius and orbit facts were taken from Wikipedia.

Planets are rendered using a simple sphere model provided for the RasterTek tutorial series, with textures compiled from the open source space game Orxonox. Various matrix transforms are performed in order to render the models in the correct locations in the world. First, the planet is scaled based on it’s radius. It is then rotated about y based on the current frame time and its rotation speed, then rotated again based on its axial tilt, this time around the z axis. It is then translated to it’s correct location in space. This location is calculated by first translating it out to it’s orbit radius, then rotating around the orbit center and y axis, then rotating again about the z axis using the orbit inclination. The result is a realistic simulation of the solar system movement, both in rotation and orbit.

Realistic lighting was achieved using a point light system centered on the sun. From this point direction both diffuse and specular lighting were calculated. A very low-level ambient light was also used in order to simulate starlight. In addition, the sun and skybox were rendered with ambient at full. Without this, the sun would not have any lighting at all since the point light comes from inside it, and the skybox would have been lit unevenly.

Camera controls are complete, allowing movement and rotation in every direction. The player uses the arrow keys to move left, right, forward, and backward, and the space and left control keys to ascend or descend. The W and S keys control pitch, A and D yaw, and Q and E roll. The game also allows the user to mute the music and sound if desired using the M key, and to turn it back on using the N key. The escape key exits the game.

FPS and CPU counters were added using windows timing functions and the performance data helper. They process data each frame, and are then rendered to the screen as a 2d texture using a font class. Other 2D texture rendering includes the title screen, control screen, HUD, and mouse cursor. All 2D textures were created by Roy Stevens and are used with permission.

The scene uses a modified skybox as a backdrop. Since the scene is in space, a simple starfield suffices for a background. To create this effect, a starfield-textured cube is rendered on top of the camera position, with Z buffer and back culling turned off. This gives the effect of the same background no matter where the camera is located, but different backgrounds based on the rotation of the camera.

Direct Sound was used to implement sound in the game. Background music is looped until the user chooses to mute it, after which the volume is set to zero. A spaceship engine sound is also looped, with the volume turned up when the camera is moving. A sliding volume effect is achieved using an update function called each frame. Music is Ignis by M-PeX. Both the music and sound effect fall under creative commons licenses and are legal to use.

The final feature in the scene is the particle effect system used for Halley's Comet. The comet tail uses a simple particle effect in which particles are dropped at a random position close to the comet center. The particles then move at a random velocity, disappearing when they get too far from the comet. Since the comet is also moving on it's own, this gives the expected tail effect. The particles are rendered using 2D billboards with alpha enabled. The sprite for these billboards comes from the RasterTek tutorial. Sorting of the particles is done via the insertion sort algorithm.

Program Structure

Code organisation maintains much of the structure provided from the in-class and RasterTek tutorials. The game is primarily managed by a SystemClass, which contains within it an InputClass, PositionClass, SoundClass, GraphicsClass, and each timer class (FPS, CPU, and Timer). These classes are separated as much as possible, with SystemClass passing data between them. For example, the SystemClass gets key presses from InputClass and passes them to PositionClass. PositionClass then modifies the camera position, which is retrieved by SystemClass and passed to GraphicsClass for use in rendering.

GraphicsClass holds an instance of all classes related to the display of the game. This includes:

- The D3DClass, which controls major DirectX objects and the graphics device information
- The CameraClass, which controls the camera and generates the view matrix
- The light, texture, and particle shaders
- Bitmaps containing the crosshairs, HUD, title screen, and controls screen
- Text class for displaying FPS and CPU counters
- PlanetClass instances for each of the astronomical objects
- A ModelClass containing the skybox
- A ParticleSystemClass to control the comet tail
- A LightClass containing the point light used to simulate the sun

The following diagram demonstrates this code structure using the UML class diagram syntax:

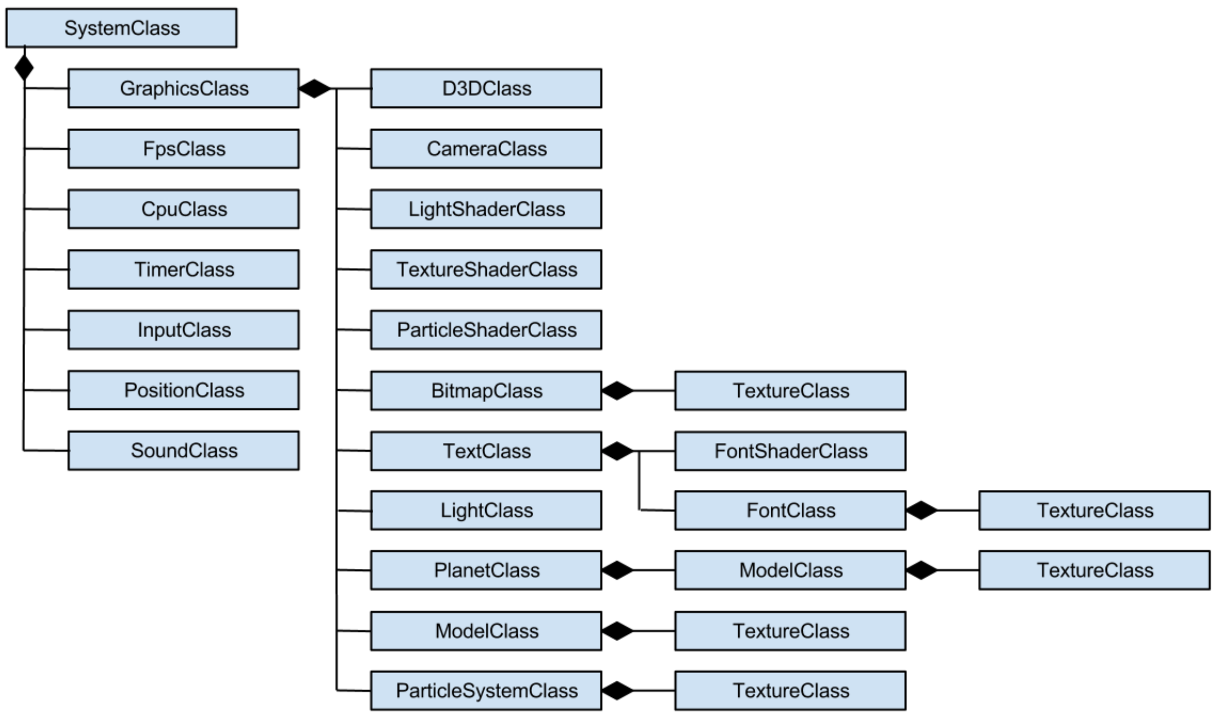

Figure 1: *Class diagram in the UML style*

Results

The program as a whole was ultimately successful in demonstrating the various capabilities of DirectX 11. However, the program structure suffered greatly due to the programmer's unfamiliarity with the content. Much of the structure was taken directly from the RasterTek tutorials, and while this structure was acceptable, it was not especially object oriented and could have taken much better advantage of inheritance and polymorphism.

Encapsulation was difficult to maintain as more functionality was added to the program. In particular, it was difficult to keep game objects separate from their corresponding graphics object. Currently the graphics class handles both updating planet locations as well as rendering their models. This could be improved by having planet classes in a separate game class, with pointers to models owned by the graphics class. The planets would only update those models, which would in turn be rendered in the correct location by the graphics class.

Had time allowed, the program structure would have been redesigned in a fashion similar to the class diagram below:

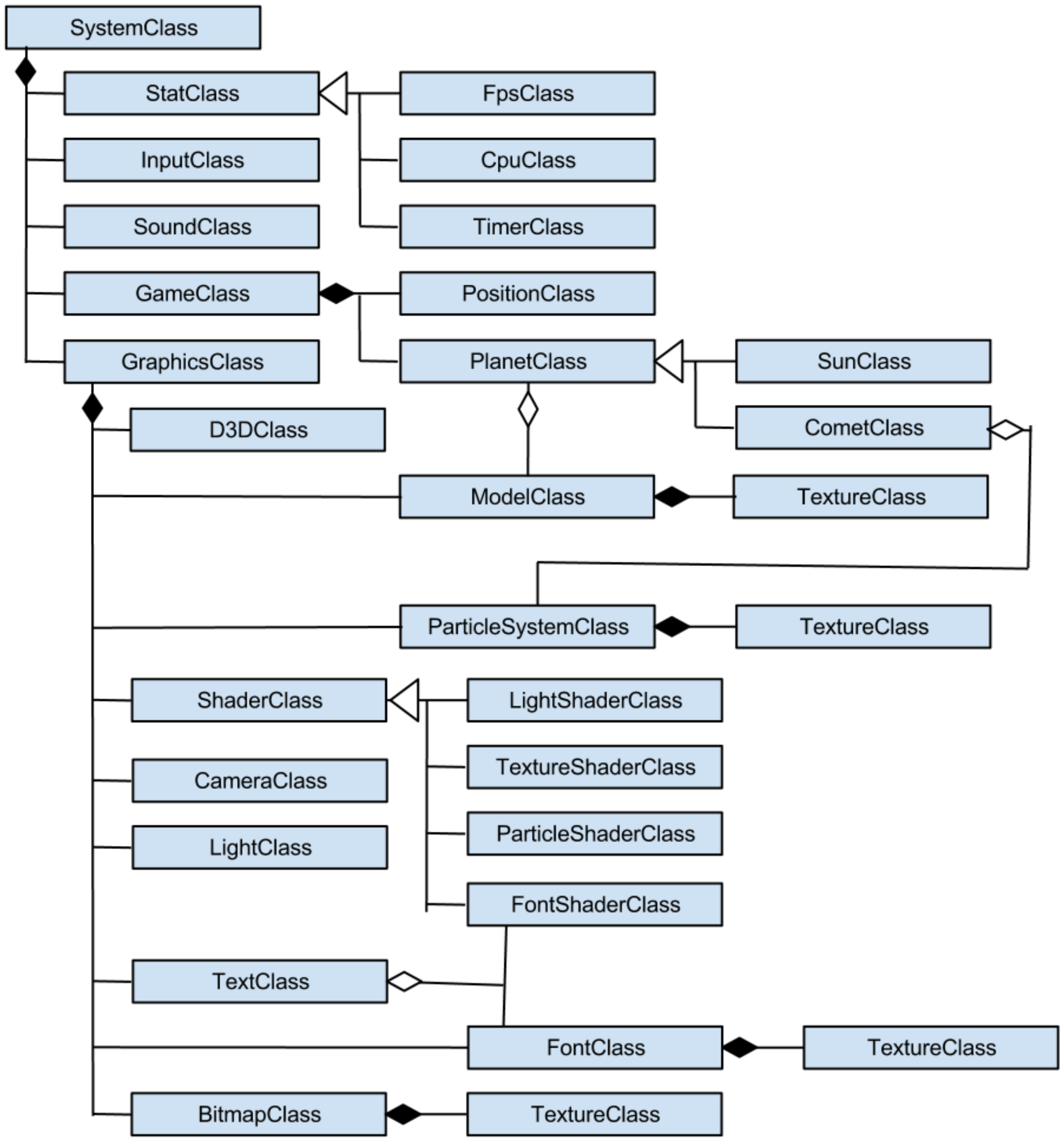

Figure 2: *Proposed class diagram in the UML style*

In this proposed version, inheritance is set up for the shaders, timers, and game objects. All shaders are owned by GraphicsClass, so TextClass only has a weak association with FontShaderClass (it has a pointer to it but doesn't control it's initialisation or shutdown). Similarly, FontClass is owned by GraphicsClass so the same font could be shared for multiple TextClass objects, so again TextClass simply has a pointer to it but does not own it. All models are owned by GraphicsClass, but the PlanetClass game object keeps a pointer to it's corresponding model so it can update the model's position in the game world for later rendering.

While this approach seems more complex, it would make much more sense in practice and make good use of inheritance, dynamic binding, and other useful object oriented features.

In terms of efficiency, the program was fairly effective. With VSync turned on, a constant 60 FPS and nearly 0% CPU usage was achieved using a high end gaming PC. However, it is expected that these figures would worsen when the program is run on a lower end PC. The program is coded to be compatible with older video cards and DirectX 10.

Conclusions

A major lesson gained from this project was that planning is key to keeping a program organized and improving encapsulation. The tutorials provided on RasterTek built a good ground work, but did not take into account a more complex program with multiple game objects. Because of this, no game class was built in to the structure, and no advice was given on keeping game logic updates separate from graphics updates.

It was also quite easy to make mistakes during debugging due to perception errors. Sometimes it is difficult to tell what is happening in a scene because the human brain interprets the scene by correlating it with what is expected - which is not always what is actually happening. A good solution is to implement camera movement as early as possible, as this will allow debugging from multiple angles, which helps avoid this issue.

Most importantly, it was very easy to get bogged down by the complexity of the Direct3D code. However, experimentation is the best way to learn what is happening in a particular piece of code, so the programmer must not be daunted by the syntax and instead simply try it out, refining along the way.

This project did an excellent job of introducing the basics and even some more advanced features of DirectX 11. Building on these foundations, a full graphics engine could be built. Additionally, a greater understanding of how graphics engines work was gained, which will aid in using graphics engines in the future.

References

Bett, M. (2012) *AG1101A Lecture Notes*, Exeter: University of Abertay, Dundee.

En.wikipedia.org (2013) *Solar System - Wikipedia, the free encyclopedia*. [online] Available at: http://en.wikipedia.org/wiki/Solar_System [Accessed: 18 Jan 2013].

Freemusicarchive.org (2013) *Free Music Archive*. [online] Available at: http://freemusicarchive.org/ [Accessed: 18 Jan 2013].

Freesound.org (2006) *Freesound.org - Freesound.org*. [online] Available at: http://www.freesound.org/ [Accessed: 18 Jan 2013].

Orxonox.net (2012) *Orxonox*. [online] Available at: http://www.orxonox.net/ [Accessed: 18 Jan 2013].

Rastertek.com (2012) *RasterTek - DirectX 10 and DirectX 11 Tutorials*. [online] Available at: http://www.rastertek.com/ [Accessed: 18 Jan 2013].