Programming for graphics

For this assignment I have been tasked with creating an application or game using various methods for generating graphics to a screen using OpenGL and GLUT. GLUT is OpenGL’s utility toolkit providing a simple standard for handling events as well as window management (Renka, 2004). In this evaluation I will discuss the range of methods I used in my game, including graphic primitives, transformations, textured meshes and lighting. I also researched alternate ways to create graphics which are; producing text, creating more complex primitives, adding multiple windows and a detailed menu system.

Due to the vast amount of graphical components, I have created two applications. The first is a basic game of ‘Pong’ that demonstrates the basic primitives, textured meshes, basic transformations, a menu system, multiple windows and text rendering, which I will refer to as App 1.

The second, App 2, is a basic window generating a cube hit by a spotlight. The application has a similar menu and clearly displays the functionality of the lighting effect.

Graphic Primitives

Graphic primitives are various lines, points and triangles that are defined as an ordered set of vertices (Astle and Hawkins, 2004). App 1 initially used basic rectangular shapes to create the paddles and the scoring system but, as the source code will show, the rectangles were changed to become cuboids. This meant changing entire projection matrix to exist within 3D space. The rectangles were changed, as I wanted to change the angle of the game using the ‘GluPerspective’ function, which specifies a viewing frustum based on the world coordinate system or the aspect ratio of the associated viewport (Microsoft, 2017). This would give the game some depth but I ran into a few problems. The biggest problem I encountered was with the design of the orthographic projection matrix. Orthographic projections are a collection of 2D graphics that attempt to accurately represent an object through a specified projection (Howarth, 1960). I designed the ortho values to fit the screen size so that the game would scale up or downwards. This meant that when trying to compensate for the change in perspective, the camera would attempt to look at an area away from the center of the screen. I also encountered a few problems whilst creating the ball. There are no clean ways to generate a circle using GLUT, so after researching the best ways to achieve this I used a triangle fan function. A triangle function uses the average position of multiple triangles created around a centre point to create a circular shape that is smoother the using other drawing techniques (Huang, Bevilacqua and Premaratne, 2014).

The cube in App 2 uses the same 3D ortho as the App1. However the application uses a zero to one ratio for the orthographical projection making the change of perspective and camera angles easier to manage, as the width and height are not relative to the window size. The cube was easier to generate than the paddles due to the x, y and z values being identical.

Transformations

The transformations on App 1 range from simple movement on the y axis to more complex movement, with the ball moving itself without user input based on collision with each paddle.

Paddle 1 uses a basic keyboard function built into OpenGL that identifies each key based on the ascii value. Paddle 2 was harder to create the traditional multiplayer keyboard controls, i.e. the up and down arrows on the directional keys as there are no ascii values for these keys. To emulate the controls I had use a separate keyboard function built into GLUT.

The Ball has it’s own velocity. This made the game feel more natural and kept a sense of immersion. The velocity is based on whether it has collided with either the top or bottom walls or with either paddles, this is so that the player could easily relate to the original Pong game.

App 2 has only one transformation that leaves a cube spinning on a pivot. I have utilised the GLUT menu command to increment and decrement the spinning speed of the cube as well as reset the speed, creating a sense of interactivity for the user.

Textured meshes

App 1 has basic examples of textured meshes. The paddles are both using the same image. This is to understand fundamentally how texturing an object works. OpenGL uses a coordinate system for each vertex of an object to bind to each vertex of the texture (Astle and Hawkins, 2004). This leads to creating multiple textures on one 3D object in App 2. The cube has six faces each of which has a separate texture bound to all the individual vertices recreating a dice.

Lighting

App 2 displays basic lighting techniques. The cube is a black and white dice that, due to the lighting effect, emits a red glow. Lighting was tough to do on this cube due to the fact that it is completely textured. During the initialisation of the texturing I had to define how the material is applied to the cube, which contained a GLUT function that replaced anything that was applied to the cube to instead overwrite it with the texture. I ended up having to use a GL\_MODULATE function that simply modified the texture to include any predefined effects. Due to this realisation, the code was easy to modify and tidy so that the lighting was simple to change if required.

Multiple windows

GLUT library uses a window creation command that is completely independent from Microsoft and Unix making it platform independant (Whitrow, 2008). This meant creating multiple windows, and having them interact with each other was relatively easy. I started by creating a simple window that displayed the controls for each player then quickly progressed to having an independant window to track the scores for each player. Window creation only supports basic events such as window resizing and keyboard and mouse input. This meant that I was limited to using keyboard controls without an ability to have multiple keystrokes. All of the window events that need handling are based on callback from specific functions (Boreskov and Shikin, 2013). The scoring system is triggered by the ball leaving either the left or right of the screen forcing a basic quad primitive to be generated on the adjoining window.

Text

When attempting to create simple text to give basic information to the user, such as providing the controls and player’s scoreboards, I found that OpenGL does not have an option to create raw text to the screen. Instead it requires a Bitmap Character function. In OpenGL text characters are represented using polygon shapes using a built in font library then simply copies the image of the character from the font cache into the frame buffer to display the letter at the desired location (Chen and Chen, 2006). As the ‘text’ is being loaded from a library there are only a few options for fonts and sizes making the text displayed look substandard but the overall result was excellent as I managed to portray the information I needed to the user.

Menu system

In both App 1 and App 2 I have utilised the GLUT library which supports rudimental pop-up menus. App 1 allows the user to change the colour scheme of the game using the aforementioned menu. The GLUT library also supports nested, or hierarchical, menus in which multiple items can contain submenus (Boreskov, 2006). This allowed me to create a broader variety of colour schemes to show a more extensive use of the menu command. App 2 uses a similar nested menu structure to allow the user to change the speeds in which to cube spins. Initially I used buttons to adjust the way the game looked, but the game quickly became unstructured and cumbersome. I decided that a menu, on right click, would be cleaner and more user friendly.

To conclude, both applications clearly display basic and advanced processes of generating graphics to a window. The applications had many problems which I have overcome such as developing a strong understanding of how the orthographical projection matrix is created, how textured meshes are plotted and how to create multiple textures and learning the extent of GLUT and what you can create using OpenGL.

Bibliography

Astle, D. and Hawkins, K. (2004). *Beginning OpenGL game programming*. 1st ed. Boston, MA: Thomson/Course Technology.

Boreskov, A. and Shikin, E. (2013). *Computer graphics*. 1st ed. CRC Press, 2013.

Boreskov, A. (2006). *Developing and debugging cross-platform shaders*. 1st ed. Wayne, Pa.: A-List.

Chen, J. and Chen, C. (2006). *Foundations of 3D graphics*. 1st ed. New York: Springer.

Huang, D., Bevilacqua, V. and Premaratne, P. (2014). *Intelligent computing theories and methodologies*. 1st ed.

Howarth, A. (1960). A BACKGROUND TO ORTHOGRAPHIC PROJECTION. *Education + Training*, 2(7), pp.24-28.

Microsoft, (2017). *gluPerspective function (Windows)*. [online] Msdn.microsoft.com. Available at: https://msdn.microsoft.com/en-us/library/windows/desktop/dd368675(v=vs.85).aspx [Accessed 8 Jan. 2017].

Renka, R. (2004). Algorithm 834. *ACM Transactions on Mathematical Software*, 30(2), pp.212-217.

Whitrow, R. (2008). *OpenGL graphics through applications*. 1st ed. New York: Springer.