

CMP203 – Graphics Programming Coursework Report

Sam Gallacher, 1700284

Brief Coursework Overview

In my coursework's scene, I have loosely recreated the interior of Moe's Tavern, the implementation of which is based on the version seen within the game The Simpsons: Hit and Run.

This was chosen as its scope isn't too great that it would become difficult to complete within the amount of time given, yet is still a decent scale and allows all of the coursework criteria to be met.

This also allowed using the interior in the game as reference, which enabled me not only to demonstrate my graphics programming skills learned in class, but also understand how games actually implement their scenes, models and textures by observing a practical real life example from a game.

This document will explain all user controls, as well as all implemented features and how each of these correlate to meeting their relative requirements from the coursework specification.

User Controls

The user controls in the application have been designed to be easy to use and to feel natural, particularly in the case of the cameras. A list of controls is below:

- | | | | |
|----------|---|-------------------------------------|---|
| 1 | - | Switch to free-roam camera | The free-roam camera is completely user controlled, allowing any combination of pitch, yaw and roll |
| 2 | - | Switch to bar strafe camera | The bar strafe camera can be controlled with either A and D or W and S depending on the orientation of the camera. It allows the user to pan between two defined positions while linearly strafing down a set axis. |
| 3 | - | Switch to rotating camera | The rotating camera is completely procedurally controlled and does not allow any user input. It allows the user to observe the rotating camera's POV of the bar as it pans and rotates between two defined positions. |
| 4 | - | Toggle First Column Lights | Toggles on/off the first column of roof lights on the left hand side. |
| 5 | - | Toggles Second Column Lights | Toggles on/off the second column of roof lights on the right hand side. |
| 6 | - | Toggle Pool Table Lights | Toggles on/off the pool table's spot lights. |
| R | - | Reset camera | Pressing R will reset the actively selected camera to its default position. |
| 0 | - | Activate Slot Machine | Pressing 9 will activate the slot machine in the corner of the bar, so that its lights and animation can be seen. |
| # | - | Toggle Wireframe Mode | |

Camera Controls

Free-Roam Camera:

W	-	Move forward
S	-	Move backward
A	-	Strafe left
D	-	Strafe right
T	-	Pitch up
G	-	Pitch down
Y	-	Yaw Right
H	-	Yaw Left
U	-	Rotate anti-clockwise
J	-	Rotate clockwise
I	-	Strafe up
K	-	Strafe down
Mouse	-	Rotate view

Strafe Camera (if applicable – depends on axis and rotation):

W	-	Strafe forward
S	-	Strafe backward
A	-	Strafe left
D	-	Strafe right

Rotating Camera – N/A, is procedurally controlled

Coursework Requirements

Lighting

Lights are heavily prevalent throughout my scene, with multiple types, colours and animation included. For instance, take a look below at the overall scene lighting and the pool table lights which are different types – they are point and spot lights respectively:

Point Lights



Spot Lights:



Lighting

I've also included lights that have different colours and animate, in the slot machine. When the slot machine finishes rotating, a light animates and flashes green to signify a win. A picture of the light illuminated is below:



Lighting

To represent my lights, I've created a base light class and made child classes for each light, with an overridden render function. This enables me to render light properties for specific types aka whether a light will have attenuation or not, whether it will have spotlight properties, etc. Then, I've made a Light Manager class to keep track of, update, render and enable/disable all of my scenes lights at will. A portion of the Light and Light Manager classes are visible below:

```
#include <glut.h>
#include <gl/gl.h>
#include <gl/glu.h>
#include <algorithm>
#include <vector>

class Light
{
public:
    Light(int id, std::vector<GLfloat> pos);
    ~Light();

    //Setters for light properties
    void SetAmbient(float r, float g, float b);
    void SetDiffuse(float r, float g, float b);
    void SetAttenuation(float constant, float linear, float quadratic);

    //Gets ID (GL_LIGHTX) of this light
    int GetID();

    //Getters and Setters
    bool GetIsOn();
    void SetIsOn(bool state);

    bool IsFlashing();

    std::vector<GLfloat> GetPosition(); //Gets where the light is in the world

    //Flash flashes the light for flashDuration, turning it on and off (1 cycle) in cycleTime seconds
    void Flash(float cycleTime, float flashDuration, float dt);

    void Update(float dt);
    virtual void Render(); //Virtual function, lights render differently based on type

    bool CurrentlyFlashing = false;
```

```

#pragma once
#include <glut.h>
#include <gl/gl.h>
#include <gl/glu.h>
#include <algorithm>
#include <vector>
#include <memory>
#include "Light.h"
#include "SpotLight.h"

class LightManager
{
public:
    LightManager();
    ~LightManager();

    void AddNewLight(Light* light); //Add a new light to the list
    void UpdateAll(float dt); //Update all the lights
    void RenderAll(); //Render all the lights

    void ToggleLight(Light* light); //Toggle light on/off

protected:
    std::vector<Light*> _Lights;
};

```

In the Light class, the Flash function is responsible for the animation. By taking a total time to flash, flashDuration, and a time of one cycle (where a cycle is defined as the time for the light to cycle on, off and then on again), cycleTime, calculations can be performed in the flash function to alter the Ambient and Diffuse values over flashDuration to animate a flashing light. Then, in Update, the Diffuse and Ambient values can be updated to animate the flash itself. The Update function is below:

```

void Light::Update(float dt)
{
    if (_ShouldFlash) //If we should flash
    {
        _DurationFlashed += dt; //Increment by delta time

        //If we've gone past the duration
        if (_DurationFlashed >= _FlashDuration)
        {
            //Don't flash anymore, reset values
            _ShouldFlash = false;
            _FlashDuration = 0.0;
            _DurationFlashed = 0.0;
            Ambient = _PreFlashAmbient;
            Diffuse = _PreFlashDiffuse;
            CurrentlyFlashing = false;
        }

        else if (shouldAdd) //If we should add the increment values, do so
        {
            Ambient = { Ambient[0] + (_FlashAmbValues[0] * dt), Ambient[1] + (_FlashAmbValues[1] * dt), Ambient[2] + (_FlashAmbValues[2] * dt) };
            Diffuse = { Diffuse[0] + (_FlashDifValues[0] * dt), Diffuse[1] + (_FlashDifValues[1] * dt), Diffuse[2] + (_FlashDifValues[2] * dt) };

            if (Ambient[0] >= _FlashAmbMax && Diffuse[0] >= _FlashDifMax) //If we've hit or gone past the maximum values we should reach during flashing
            {
                shouldAdd = false; //We should begin subtracting the increment value
            }
        }

        else if (!shouldAdd) //If we should subtract the increment values, do so
        {
            Ambient = { Ambient[0] - (_FlashAmbValues[0] * dt), Ambient[1] - (_FlashAmbValues[1] * dt), Ambient[2] - (_FlashAmbValues[2] * dt) };
            Diffuse = { Diffuse[0] - (_FlashDifValues[0] * dt), Diffuse[1] - (_FlashDifValues[1] * dt), Diffuse[2] - (_FlashDifValues[2] * dt) };

            if (Ambient[0] <= 0.0f && Diffuse[0] <= 0.0f) //If we get to or below 0
            {
                shouldAdd = true; //Begin adding the increment values again
            }
        }
    }
}

```

Camera and Interaction

I've implemented several cameras of different types and one that is procedurally controlled using a parent Camera class I've developed and extended from the lab tasks. From the base camera class, I've created child classes for each camera type. The camera types implemented include a free-roam camera, a strafing camera and an auto rotating camera. The auto rotating camera is procedurally controlled – by passing a speed, a minimum and maximum value, the camera will rotate between the two values at the set speed. Then, by using a camera pointer, I can easily switch between them in the scene and update them as each camera requires, by using virtual functions that I can override. Below is a small snippet of the base Camera class header:

```
public:
    Camera(float speed, Vector3 origin, Vector3 pitchYawRoll);
    ~Camera();
    void Update();
    void Reset();
    void SetBounds(Vector3 minBounds, Vector3 maxBounds);
    void SetOrigins(Vector3 origin, Vector3 pitchYawRoll);

    virtual void Animate(float dt);
    virtual void MoveForward(float dt);
    virtual void MoveRight(float dt);
    virtual void MoveUp(float dt);
    virtual void RotateX(float dt);
    virtual void RotateY(float dt);
    virtual void RotateZ(float dt);

    virtual void RotateFromMouse(float xOffset, float yOffset, float dt);
```

Then, by overriding the Move functions, I can limit what each camera is capable of in terms of movement. For example, the strafe camera does not allow any form of rotation, so the overrides are empty as below:

```
void StrafeCamera::RotateX(float dt)
{
}

void StrafeCamera::RotateY(float dt)
{
}

void StrafeCamera::RotateZ(float dt)
{
}

void StrafeCamera::RotateFromMouse(float xOffset, float yOffset, float dt)
{
}
```

This logic is also applied to the procedurally controlled rotating camera – all movement functions are overridden to be empty so the user cannot move the camera.

(It is important to note that the user can interact with other elements of the scene than just the camera. They can toggle the lights in the scene, as well as use the slot machine – these will be demonstrated and mentioned throughout this report)

I feel it's pertinent to bring the SetBounds function and Animate functions to attention. Using SetBounds, I can specify a minimum and maximum point in 3D space between which the camera is allowed to move. This enables me to “lock” the user into my scene's room and make it impossible to go outwith my intended viewing area. The Animate function is also useful as it can be overridden by any camera type that may require it and perform any operation needed. This was used in the procedurally controlled rotating camera as below:

```

void SelfRotatingCamera::Animate(float dt)
{
    if (!_IsPaused && (_Yaw > _MaximumRot || _Yaw < _MinimumRot))
    {
        _IsPaused = true;
        _Yaw -= _Speed * dt;    //Prevents camera getting stuck, good cheat, unnoticable
    }

    else if (!_IsPaused)
    {
        switch (_RotationAxis)
        {
            case RotationAxis::Y:
                _Yaw += _Speed * dt;
                break;

                //This could be expanded in the future to add other axis's of rotation if needed
        }
    }

    if (_IsPaused)
    {
        _TimePaused += dt;
        if (_TimePaused >= _PauseBoundsDuration)
        {
            _TimePaused = 0.0;
            _Speed = -_Speed;
            _IsPaused = false;
        }
    }

    Update();
}

```

Basic Geometry

Several hand-made geometries are present in my scene, the most common being the wall posters visible on the walls of my scene, as seen here being correctly textured and lit:



```
#pragma region TallPoster
glBindTexture(GL_TEXTURE_2D, duffTallTexture);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);

glBegin(GL_QUADS);
    glTexCoord2f(0.0f, 0.0f);
    glNormal3f(0.0f, 0.0f, 1.0f);
    glVertex3f(2.0f, 3.0f, -15.9f);
    glTexCoord2f(0.0f, 1.0f);
    glNormal3f(0.0f, 0.0f, 1.0f);
    glVertex3f(2.0f, 1.5f, -15.9f);
    glTexCoord2f(1.0f, 1.0f);
    glNormal3f(0.0f, 0.0f, 1.0f);
    glVertex3f(3.0f, 1.5f, -15.9f);
    glTexCoord2f(1.0f, 0.0f);
    glNormal3f(0.0f, 0.0f, 1.0f);
    glVertex3f(3.0f, 3.0f, -15.9f);
glEnd();
#pragma endregion
```


Basic Geometry

For an extra element of complexity, I've added shapes other than quads. In the case of the flag, I've used a triangle as can be seen below:



Basic Geometry

I've also included transparency in my scene, in the form of a simple glass pane cabinet behind the bar.



I've also used transparency again and depth sorting with my glass table which can be observed in the Shadows and Stencil Buffer section later on in the report.

Hierarchical Modelling

Hierarchical Modelling and use of the matrix stack is prevalent heavily throughout my scene, but the most profound example which also includes hierarchical animation is the slot machine. The slot machine uses the matrix stack to translate its child “wheel spinners” to their relative positions and animate the rotation when the machine is activated. This code sample demonstrates this:

```
void SlotMachine::render(Vector3 position)
{
    glPushMatrix();
    glTranslatef(position.x, position.y, position.z);
    glRotatef(-90.0f, 0.0f, 1.0f, 0.0f);
    Model::render();

    glPushMatrix();
    glRotatef(30.0f, 1.0f, 0.0f, 0.0f);
    glTranslatef(0.0f, 0.0f, -0.1f);
    glScalef(1.0f, 0.8f, 0.8f);

    glPushMatrix();
    glTranslatef(-0.28f, 0.0f, 0.0f);

    if (_Rotation <= 360.f * 4.0f)
    {
        glRotatef(_Rotation, 1.0f, 0.0f, 0.0f);
    }

    _SlotWheels[0].render();
    glPopMatrix();

    glPushMatrix();

    if (_Rotation <= 360.f * 5.0f)
    {
        glRotatef(_Rotation, 1.0f, 0.0f, 0.0f);
    }

    _SlotWheels[1].render();
    glPopMatrix();
}
```

Geometry Generation

The Pool Table not only has a further use of hierarchical modelling with the pool balls, but the balls themselves are procedurally generated spheres. Using a base shape class as a parent and creating a child Sphere class, I was able to develop a function that'd draw a sphere of a given radius with a given amount of quads used to implement the latitude and longitudes of the sphere. To generate the 15 pool balls on the table, I simply populate an array with them, setting the appropriate texture for each, with a latitude and longitude quad count of 25 and a radius of 0.1. In code, vertex, normal and texture coordinate arrays are filled to be used with the rendering code. I then perform another use of hierarchical modelling and render the balls in their appropriate translated position. The following shows the sphere generation code itself, the use of the Sphere constructor to fill the array and the rendering code:

Sphere Generation

```
void Sphere::Generate(GLuint texture, float lat, float lon, float r)
{
    std::list<Vector3> verticesList;
    std::list<Vector3> normalsList;
    std::list<float> texCoordsList;
    _Texture = texture;

    float theta = (2.0f * PI) / lat;
    float delta = (PI) / lon;

    for (float i = 0; i < lon; i++)
    {
        for (float j = 0; j < lat; j++)
        {
            Vector3 topLeft = Vector3((r * cosf(theta * i)) * sinf(delta * j), r * cosf(delta * j), ((r * sinf(theta * i)) * sinf(delta * j)));
            Vector3 bottomLeft = Vector3((r * cosf(theta * i)) * sinf(delta * (j + 1.0f)), r * cosf(delta * (j + 1.0f)), ((r * sinf(theta * i)) * sinf(delta * (j + 1.0f))));
            Vector3 bottomRight = Vector3((r * cosf(theta * (i + 1.0f))) * sinf(delta * (j + 1.0f)), r * cosf(delta * (j + 1.0f)), ((r * sinf(theta * (i + 1.0f))) * sinf(delta * (j + 1.0f))));
            Vector3 topRight = Vector3((r * cosf(theta * (i + 1.0f))) * sinf(delta * j), r * cosf(delta * j), ((r * sinf(theta * (i + 1.0f))) * sinf(delta * j)));

            _Normals.push_back(topLeft.x / r);
            _Normals.push_back(topLeft.y / r);
            _Normals.push_back(topLeft.z / r);
            _TextureCoords.push_back(1 - (i / lon));
            _TextureCoords.push_back(1.0f - (j / lat));
            _Vertices.push_back(topLeft.x);
            _Vertices.push_back(topLeft.y);
            _Vertices.push_back(topLeft.z);

            _Normals.push_back(topRight.x / r);
            _Normals.push_back(topRight.y / r);
            _Normals.push_back(topRight.z / r);
            _TextureCoords.push_back(1 - ((i + 1) / lon));
            _TextureCoords.push_back(1.0f - (j / lat));
            _Vertices.push_back(topRight.x);
            _Vertices.push_back(topRight.y);
            _Vertices.push_back(topRight.z);

            _Normals.push_back(bottomRight.x / r);
            _Normals.push_back(bottomRight.y / r);
            _Normals.push_back(bottomRight.z / r);
```

```
_TextureCoords.push_back(1 - ((i + 1) / lon));  
_TextureCoords.push_back(1.0f - (j + 1) / lat);  
_Vertices.push_back(bottomRight.x);  
_Vertices.push_back(bottomRight.y);  
_Vertices.push_back(bottomRight.z);  
  
_Normals.push_back(bottomLeft.x / r);  
_Normals.push_back(bottomLeft.y / r);  
_Normals.push_back(bottomLeft.z / r);  
_TextureCoords.push_back(1 - (i / lon));  
_TextureCoords.push_back(1.0f - ((j + 1) / lat));  
_Vertices.push_back(bottomLeft.x);  
_Vertices.push_back(bottomLeft.y);  
_Vertices.push_back(bottomLeft.z);
```

Sphere Array Filling

```
for (int i = 0; i < 15; i++)  
{  
    _PoolBalls[i] = Sphere(loadTextureRet("Assets/Textures/Pool Balls/Ball" + std::to_string(i + 1) + ".png"), 25, 25, 0.1);  
}
```

Sphere Rendering

```
void PoolTable::render(Vector3 position)
{
    glPushMatrix();
    glTranslatef(position.x, position.y, position.z);

    glPushMatrix();
    glTranslatef(0.0f, 0.35f, -0.70f);
    glScalef(0.65f, 0.65f, 0.65f);

    glPushMatrix();
    _PoolBalls[0].Draw();

    glPushMatrix();
    glTranslatef(0.0, 0.0f, -0.15f);

    glPushMatrix();
    glTranslatef(0.1f, 0.0f, 0.0f);
    _PoolBalls[1].Draw();
    glPopMatrix();

    glPushMatrix();
    glTranslatef(-0.1f, 0.0f, 0.0f);
    _PoolBalls[2].Draw();
    glPopMatrix();

    glPushMatrix();
    glTranslatef(0.0, 0.0f, -0.15f);

    glPushMatrix();
    glTranslatef(0.2f, 0.0f, 0.0f);
    _PoolBalls[3].Draw();
    glPopMatrix();

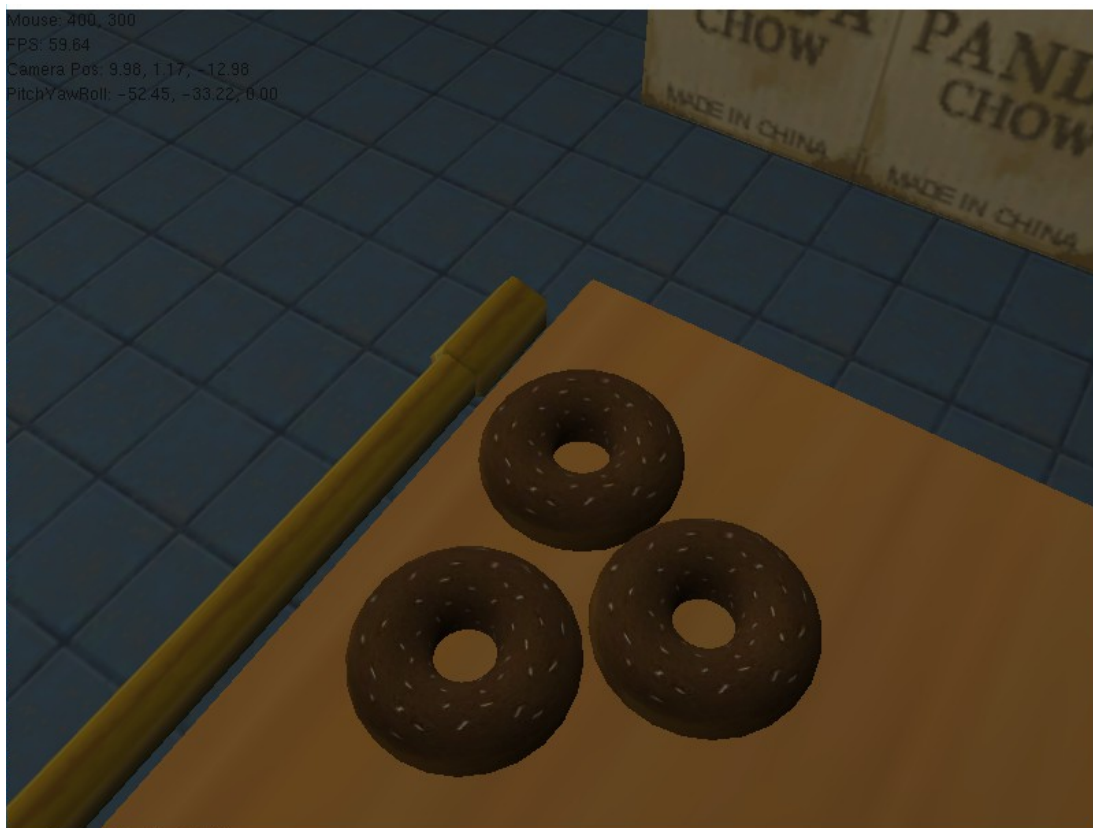
    _PoolBalls[7].Draw();

    glPushMatrix();
    glTranslatef(-0.2f, 0.0f, 0.0f);
    _PoolBalls[5].Draw();
    glPopMatrix();
}
```


Result



Furthermore, I've also added a procedurally generated shape that is more complex than what was covered in class – a Torus. I've then placed several of them on the bar and textured them to appear as donuts, with the result visible here:



Torus Generation

```
void Torus::Generate(GLuint texture, int segs, double internalRad, double bodyRad)
{
    _Texture = texture;

    double internalAngle = 0.0f;
    double externalAngle = 0.0f;

    double theta = 2 * PI / segs;
    double delta = 2 * PI / segs;

    for (double i = 0; i < segs; i++)
    {
        glBegin(GL_QUADS);
        for (double j = 0; j < segs; j++)
        {
            //Vertex calculations
            Vector3 first = Vector3
            (
                (internalRad + bodyRad * cosf(internalAngle))*cosf(externalAngle),
                (internalRad + bodyRad * cosf(internalAngle))*sinf(externalAngle),
                (bodyRad*sinf(internalAngle))
            );

            Vector3 second = Vector3
            (
                (internalRad + bodyRad * cosf(internalAngle))*cosf(externalAngle + delta),
                (internalRad + bodyRad * cosf(internalAngle))*sinf(externalAngle + delta),
                (bodyRad*sinf(internalAngle))
            );

            Vector3 third = Vector3
            (
                (internalRad + bodyRad * cosf(internalAngle + theta))*cosf(externalAngle + delta),
                (internalRad + bodyRad * cosf(internalAngle + theta))*sinf(externalAngle + delta),
                (bodyRad*sinf(internalAngle + theta))
            );
```

```

Vector3 fourth = Vector3
(
    (internalRad + bodyRad * cosf(internalAngle + theta))*cosf(externalAngle),
    (internalRad + bodyRad * cosf(internalAngle + theta))*sinf(externalAngle),
    (bodyRad*sinf(internalAngle + theta))
);

//First vertex
//Normal calculation
Vector3 firstNormVector = Vector3(((internalRad + bodyRad * cosf(internalAngle)) * cosf(externalAngle)) - (internalRad * cos(externalAngle)),
    ((internalRad + bodyRad * cosf(internalAngle)) * sinf(externalAngle)) - (internalRad * sinf(externalAngle)),
    (bodyRad * sinf(internalAngle)));
firstNormVector.normalise();

_Normals.push_back(firstNormVector.x);
_Normals.push_back(firstNormVector.y);
_Normals.push_back(firstNormVector.z);

_TextureCoords.push_back(i / segs);
_TextureCoords.push_back(j / segs);

_Vertices.push_back(first.x);
_Vertices.push_back(first.y);
_Vertices.push_back(first.z);

//Second vertex
//Normal calculation
Vector3 secondNormVector = Vector3(((internalRad + bodyRad * cosf(internalAngle)) * cosf(externalAngle + delta)) - (internalRad * cos(externalAngle + delta)),
    ((internalRad + bodyRad * cosf(internalAngle)) * sinf(externalAngle + delta)) - (internalRad * sinf(externalAngle + delta)),
    (bodyRad * sinf(internalAngle)));
secondNormVector.normalise();

_Normals.push_back(secondNormVector.x);
_Normals.push_back(secondNormVector.y);
_Normals.push_back(secondNormVector.z);

```

```

_TextureCoords.push_back((i + 1.0f) / segs);
_TextureCoords.push_back(j / segs);

_Vertices.push_back(second.x);
_Vertices.push_back(second.y);
_Vertices.push_back(second.z);

//Third vertex
//Normal calculation
Vector3 thirdNormVector = Vector3(((internalRad + bodyRad * cosf(internalAngle + theta)) * cosf(externalAngle + delta)) - (internalRad * cos(externalAngle + delta)),
    ((internalRad + bodyRad * cosf(internalAngle + theta)) * sinf(externalAngle + delta)) - (internalRad * sinf(externalAngle + delta)),
    (bodyRad * sinf(internalAngle + theta)));
thirdNormVector.normalise();

_Normals.push_back(thirdNormVector.x);
_Normals.push_back(thirdNormVector.y);
_Normals.push_back(thirdNormVector.z);

_TextureCoords.push_back((i + 1.0f) / segs);
_TextureCoords.push_back((j + 1.0f) / segs);

_Vertices.push_back(third.x);
_Vertices.push_back(third.y);
_Vertices.push_back(third.z);

//Fourth vertex
//Normal calculation
Vector3 fourthNormVector = Vector3(((internalRad + bodyRad * cosf(internalAngle + theta)) * cosf(externalAngle)) - (internalRad * cos(externalAngle)),
    ((internalRad + bodyRad * cosf(internalAngle + theta)) * sinf(externalAngle)) - (internalRad * sinf(externalAngle)),
    (bodyRad * sinf(internalAngle + theta)));
fourthNormVector.normalise();

_Normals.push_back(fourthNormVector.x);
_Normals.push_back(fourthNormVector.y);
_Normals.push_back(fourthNormVector.z);

```

```

        _TextureCoords.push_back(i / segs);
        _TextureCoords.push_back((j + 1.0f) / segs);

        _Vertices.push_back(fourth.x);
        _Vertices.push_back(fourth.y);
        _Vertices.push_back(fourth.z);

        internalAngle += theta;
    }

    glEnd();
    externalAngle += delta;
}
}

```

Shape rendering code

```
void Shape::Draw()
{
    //Bind texture and initial vertice vector drawing code
    glBindTexture(GL_TEXTURE_2D, _Texture);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
    glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);

    glEnableClientState(GL_VERTEX_ARRAY);
    glEnableClientState(GL_NORMAL_ARRAY);
    glEnableClientState(GL_TEXTURE_COORD_ARRAY);

    glVertexPointer(3, GL_FLOAT, 0, &_Vertices.front());
    glNormalPointer(GL_FLOAT, 0, &_Normals.front());
    glTexCoordPointer(2, GL_FLOAT, 0, &_TextureCoords.front());

    glDrawArrays(GL_QUADS, 0, _Vertices.size() / 3.0f); //Method 2
    glDisableClientState(GL_VERTEX_ARRAY);
    glDisableClientState(GL_NORMAL_ARRAY);
    glDisableClientState(GL_TEXTURE_COORD_ARRAY);
    glBindTexture(GL_TEXTURE_2D, NULL);
}
```

Geometry Storage

Geometry Storage is an integral part of my scene. I've included plenty of models, and used MTL parsing to include multiple textures on an individual model. By extending the Model class from the labs, I've added extra functionality to parse MTL files, store and load the textures specified within them and then load the model's OBJ, store the OBJ data in vertex arrays and then apply the loaded textures to them correctly. Take a look at the Jukebox for example, it uses multiple textures for the wood, glass and control panel:



The loadMTL function looks like this – it sets a usingMTL flag to true so the render function knows how to handle the vertex data, allows specification of which shape was used to represent the model (typically triangles or quads), parses the specified MTL file and then loads all the associated textures:

```
bool Model::loadMTL(char* modelFilename, char* mtlFilename, int renderingShape)
{
    usingMTL = true;
    _RenderingShape = renderingShape;
    bool result;
    parseMTL(mtlFilename); //Parse the MTL file passed in

    // Load in the model data,
    result = loadModel(modelFilename);
    if (!result) //Check if model loaded correctly
    {
        MessageBox(NULL, (LPCSTR)"Model failed to load", (LPCSTR)"Error", MB_OK);
        return false;
    }

    loadAllTextures(); //Load all the textures specified from the MTL file
    return true;
}
```

The bulk of the MTL parsing code:

```
if (strcmp(lineHeader, "newmtl") == 0) //If we've found an entry
{
    //Read the ident
    char materialIdent[256];
    fscanf(file, "%s", &materialIdent);

    int i = 0;
    for (int i = 0; i < 11; i++)
    {
        char line[256];
        fgets(line, 256, file);
    }

    //Read the texture header
    char textureLineHeader[256];
    fscanf(file, "%s", &textureLineHeader);

    if (strcmp(textureLineHeader, "map_Ka") == 0)
    {
        char materialTexture[256];
        fscanf(file, "%s", materialTexture);

        //Use delims to cut path down to be relative and add to map
        std::set<char> delims{ '\\ ' };
        std::experimental::filesystem::path p(materialTexture);
        mtl(materialIdent) = p.stem().string() + ".png";
    }
}
```

The vertex array code:

```
for (int i = 0; i < faces.size(); i += 3)
{
    //Fill arrays
    vertex.push_back(verts[faces[i] - 1].x);
    vertex.push_back(verts[faces[i] - 1].y);
    vertex.push_back(verts[faces[i] - 1].z);

    texCoords.push_back(texCs[faces[i + 1] - 1].x);
    texCoords.push_back(texCs[faces[i + 1] - 1].y);

    normals.push_back(norms[faces[i + 2] - 1].x);
    normals.push_back(norms[faces[i + 2] - 1].y);
    normals.push_back(norms[faces[i + 2] - 1].z);

    if (usingMTL)
    {
        vertexTextures.push_back(facesTextures[i]);
    }
}
```

And finally the rendering code:

```
//If we're using triangles
if (_RenderingShape == GL_TRIANGLES)
{
    //3 sides, divide and increment by 3
    for (int i = 0; i + 2 <= vertex.size() / 3.0f; i += 3)
    {
        if (usingMTL) //Per-vertex textures are only applicable when using MTL
        {
            if (vertexTextures[i] != previousTexture) //If this is a different texture, rebind it
            {
                previousTexture = vertexTextures[i];
                glBindTexture(GL_TEXTURE_2D, textures[vertexTextures[i]]);
                glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
                glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
            }
        }

        //Draw vertices to make up triangle
        glBegin(GL_TRIANGLES);
        glVertex(i);
        glVertex(i + 1);
        glVertex(i + 2);
        glEnd();
    }
}
```

Advanced: Shadows and Stencil Buffer

Finally, I've used a complicated planar shadow in several places in my scene – the pool table and the jukebox. I've also added an extra layer of complexity by linking them to my lights, so that when the lights are disabled, the shadows disappear. Below are two screenshots of the shadows:





Below is the code for rendering the Jukebox with shadows. Visible is the code that handles the light logic.

```

//Clear the shadow matrix
std::fill(std::begin(shadowMatrix), std::end(shadowMatrix), 0.0);

//Generate shadow matrix for lights that are on
if (_PointOne->GetIsOn())
{
    Shadow::generateShadowMatrix(shadowMatrix, &_PointOne->GetPosition()[0], floorVerts);
}

if (_PointTwo->GetIsOn())
{
    Shadow::generateShadowMatrix(shadowMatrix, &_PointTwo->GetPosition()[0], floorVerts);
}

glDisable(GL_LIGHTING);
glDisable(GL_TEXTURE_2D);

//Shadow of model
glPushMatrix();
glColor3f(0.1f, 0.1f, 0.1f); // Shadow's colour
glPushMatrix();
    glMultMatrixf((GLfloat *)shadowMatrix);
    glTranslatef(position.x, position.y, position.z);
    Model::render();
glPopMatrix();
glPopMatrix();

glEnable(GL_LIGHTING);
glEnable(GL_TEXTURE_2D);

//Actual model
glPushMatrix();
    glTranslatef(position.x, position.y, position.z);
    Model::render();
glPopMatrix();

```

Other advanced features are present in my scene, such as the use of the stencil buffer, depth sorting and reflection. I've used these with the glass table in the corner of my scene to demonstrate reflection. The final result looks like the following:



The following code handles the drawing of the table. Visible is the user of the stencil buffer to setup the fake object to handle the reflection, the use of depth sorting to ensure the drawing order is correct and always shown on top and back face culling so that the reflected object is not visible when you look below the table.

```
//Stencil setup
glColorMask(GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE);
glEnable(GL_STENCIL_TEST);
glStencilFunc(GL_ALWAYS, 1, 1);
glStencilOp(GL_KEEP, GL_KEEP, GL_REPLACE);

glDisable(GL_DEPTH_TEST);
glEnable(GL_CULL_FACE);
glCullFace(GL_BACK);

glBegin(GL_QUADS);
    glNormal3f(0.0f, 1.0f, 0.0f);
    glVertex3f(1.0f, 1.3f, -1.0f);
    glNormal3f(0.0f, 1.0f, 0.0f);
    glVertex3f(2.0f, 1.3f, -1.0f);
    glNormal3f(0.0f, 1.0f, 0.0f);
    glVertex3f(2.0f, 1.3f, -4.0f);
    glNormal3f(0.0f, 1.0f, 0.0f);
    glVertex3f(1.0f, 1.3f, -4.0f);
glEnd();

glDisable(GL_CULL_FACE);

//"Fake" reflected model
glEnable(GL_DEPTH_TEST);
glColorMask(GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE);
glStencilFunc(GL_EQUAL, 1, 1);
glStencilOp(GL_KEEP, GL_KEEP, GL_KEEP);

glPushMatrix();
    glScalef(0.2f, -0.2f, 0.2f);
    glTranslatef(0.0f, 1.45f, 0.0f);
    colaCan.render(Vector3(1.5f / 0.2f, 1.45f / -0.2f, -3.0f / 0.2f));
    colaCan.render(Vector3(1.5f / 0.2f, 1.45f / -0.2f, -2.5f / 0.2f));
    colaCan.render(Vector3(1.5f / 0.2f, 1.45f / -0.2f, -2.0f / 0.2f));
glPopMatrix();

glDisable(GL_STENCIL_TEST);
```

```

glPushMatrix();
    glScalef(0.2f, 0.2f, 0.2f);
    colaCan.render(Vector3(1.5f / 0.2f, 1.45f / 0.2f, -3.0f / 0.2f));
    colaCan.render(Vector3(1.5f / 0.2f, 1.45f / 0.2f, -2.5f / 0.2f));
    colaCan.render(Vector3(1.5f / 0.2f, 1.45f / 0.2f, -2.0f / 0.2f));
glPopMatrix();

//Real model
glEnable(GL_BLEND);
glDisable(GL_LIGHTING);

glPushMatrix();
    glColor4f(0.3f, 0.3f, 0.9f, 0.8f);
    glBegin(GL_QUADS);
        glNormal3f(0.0f, 1.0f, 0.0f);
        glVertex3f(1.0f, 1.3f, -1.0f);
        glNormal3f(0.0f, 1.0f, 0.0f);
        glVertex3f(2.0f, 1.3f, -1.0f);
        glNormal3f(0.0f, 1.0f, 0.0f);
        glVertex3f(2.0f, 1.3f, -4.0f);
        glNormal3f(0.0f, 1.0f, 0.0f);
        glVertex3f(1.0f, 1.3f, -4.0f);
    glEnd();

    glEnable(GL_LIGHTING);
    glDisable(GL_BLEND);

glPopMatrix();

```



References

All Assets used, such as Models and Textures have been extracted (and edited as necessary) from The Simpsons Hit and Run, Vivendi Universal Games, 2003.

Numerous tutorials were used and consulted from:

Stack Overflow – <https://stackoverflow.com/>

LearnOpenGL – <https://learnopengl.com/>

OpenGL.org -

https://www.opengl.org/discussion_boards/