S1313540 – Games Programming 3 CW 17/18

Code Explanation

MAIN PAGE

For the code explanation, I will refer to the brief and in each section, explain the relevant code that fulfills the corresponding part of the marking scheme.

Github - https://github.com/KAGSme/GamesProgramming3-Coursework-2017/tree/GP3-Submission

Youtube Footage - https://www.youtube.com/watch?v=LkdeyV9O-_A&feature=youtu.be

Main Tasks

- Models – Done.
- Sound (effects and theme music) – Done.
- Lights – Done.
- Cameras – Done.
- Textures – Done.
- Collision Detection – Done.
- Keyboard toggles – Done.
- Keyboard Control of model – Done.
- Overlays – Fonts only, no other UI elements.

Extension Material

- Fog – Done.
- Shadow Mapping – Done.
- Gamepad Support – Done.
- Custom Shaders – Done.

I confirm that the code contained in this file (other than that provided or authorised) is all my own work and has not been submitted elsewhere in fulfilment of this or any other award.

Kieran Gallagher

S1313540

Computer Games(Software Development) 4th year

NOTES ABOUT THE GAME

Controls:

- Tab, f, gamepad y: switch to free camera.
- C, gamepad left bumper: switch third person and first person camera.
- Left, right arrows; gamepad left stick: move car left and right.
- X, gamepad b: mute/unmute audio.
- W,a,s,d; gamepad left stick: move free camera.
- Mouse, gamepad right stick: rotate free camera.
- Q,e,gamepad rb, gamepad lb: hover free camera up and down.
- F1: switch between lit and unlit render modes.
- R, gamepad back/select: restart game.

Might have to rebuild alut library and add the .lib to the dependencies>lib folder and the .dll into the folder that contains the produced .exe.
https://drive.google.com/open?id=1UBzfnGT3f6jl_f_Zy9m0GIm9AaYtue4l The vs solution file can be found in the alut>admin>VisualStudioDotNET.

MODELS, TRANSFORMS AND SHADERS

See Model.h, Model.cpp, Renderer.h and Renderer.cpp

I should take some time to go over some of the relationship between classes as far as general game architecture is concerned. Our game level is made up of game objects, all game objects have: a renderer(optional), a transform, a light(optional), and other components that govern the game objects behavior. Renderer objects track the model, textures and shader program used when rendering the game object.

For this game I rely on Autodesk's FBX loader library.

I combine all meshes in the scene stored in the .fbx file into one mesh when loading in the model. While this can simplify things for when I load our game level as I assign textures manually, it can obviously cause issues for models that are made up of various meshes (for example: the cars – separate chassis and wheel meshes) where each mesh should have a different material and texture. I also retrieve/generate binormals and tangents for normal mapping purposes for models with normal maps.

I rely on a separate gamedata file(xml file parsed by the library tinyxml2) to track what models get loaded, for out game that file is mainWithRoad.scn (assets/gamedata folder) and the Resource manager class stores the models in a map for easy access from other classes. Models loaded and used by the game can be viewed below.

```
<ModelList>
    <Model>sphere.fbx</Model>
    <Model>cone.fbx</Model>
    <Model>Plane.FBX</Model>
    <Model>carSUV.fbx</Model>
    <Model>RaceC.FBX</Model>
    <Model>RegularC.FBX</Model>
    <Model>coin/coin.fbx</Model>
</ModelList>
```

Models are only loaded once from the list above.

The actual rendering of the models take place in the renderer.cpp between line 67 and line 89.

It's worth noting that when rendering the models I write to 3 frame buffers(base color, normal, world position) that are later combined in lighting passes and a final deferred render pass where directional lighting is applied.

A lot of gameobjects rely on the same uniforms when their models are rendered so the GameObject.cpp render(camera * cam) handles setting up appropriate uniforms for material colors, model view projection matrices and so on.

Our Transform system is kept away from models in general as a model can be reused for multiple game objects. Transform.cpp contains all definitions for the transformation class that tracks a gameobjects position, rotation and scale in 3-dimensional space. Our rotation system relies on quaternions and http://www.opengl-tutorial.org/intermediate-tutorials/tutorial-17-quaternions/ was a resource used heavily to help us understand them and code from

http://www.euclideanspace.com/maths/geometry/rotations/conversions/quaternionToEuler/ was adapted to help with converting quaternions to Euler in the method Transform::GetRotationEuler() .

Shader loading/compilation can be viewed in the Shader.cpp and the shader program linkage can be viewed in ShaderProgram.cpp. Shaders are loaded and assigned to gameobjects in the same gamedata file mentioned previously. All shader programs are kept in the asset folder.

```
<ShaderList>
    <Shader vert="specularVS.glsl" frag="specularFS.glsl">Specular</Shader>
    <Shader vert="diffuseNormalSpecMapVS.glsl" frag="diffuseNormalSpecMapFS.glsl">diffuseNormalSpecMap</Shader>
    <Shader vert="postProcVS.glsl" frag="colorCorr1FS.glsl">PostProcess1</Shader>
    <Shader vert="postProcVS.glsl" frag="colorCorr2FS.glsl">PostProcess2</Shader>
    <Shader vert="pointLightVS.glsl" frag="pointLightFS.glsl">PointLight</Shader>
    <Shader vert="spotLightVS.glsl" frag="spotLightFS.glsl">spotLight</Shader>
    <Shader vert="skyboxVS.glsl" frag="skyboxFS.glsl">SkyBox</Shader>
    <Shader vert="simpleDepthVS.glsl" frag="simpleDepthFS.glsl">SimpleDepth</Shader>
    <Shader vert="carSUVVS.glsl" frag="carSUVFS.glsl">carSUV</Shader>
    <Shader vert="roadVS.glsl" frag="roadFS.glsl">road</Shader>
    <Shader vert="coinVS.glsl" frag="coinFS.glsl">coin</Shader>
</ShaderList>
```

Excerpt from mainWithRoad.scn

Most of our shaders make use of the blinn-phong shading model, and if the models come with it, we allow for the use normal and specular maps. The lighting calculations can be seen in the DefRendFS.glsl , PointLightFS.glsl, SpotLightFS.glsl. Shaders like diffuseNormalSpecMap only pass normal, position, color, shadow and specular data to the appropriate frambuffer that was set to the render targets for geometry render stage.

```
float visibility = ShadowCalculation(shadowCoordsOut);

FragColor = vec4(texture(texture0, vertexTexCoordsOut).rgb, texture(texture2, vertexTexCoordsOut).r * specularIntensity);

Position.a = (specularPower); //specular

Normal = vec4(worldNormal, visibility);

Position.xyz = fragPosOut;
```

Excerpt from the diffuseNormalSpecMapFS.glsl (used to render the geometry of the models like the player car)

There are a few unique shaders that do things slightly differently based on the objects requirements like the road with has it's texture coordinates panned to simulate forward movement for the player.

```
vec2 newVertexTexCoordsOut = vec2(vertexTexCoordsOut.x - (aliveTime * 3), vertexTexCoordsOut.y);
```

Excerpt from roadFS.glsl

Since certain objects have different requirements for things like shininess, and base color(if there were no texture) we do have a material struct that gameObjects make use of(see material.h). The values of the materials are then passed as uniforms in the GameObject's Render(Camera* cam) function. Material values of gameobjects are retrieved when the scene is loaded from the .scn file mentioned previously.

```
<GameObject name="PlayerCar" hasRenderer="true" texture="Car/RaceC_red_diffuse.png,Car/RaceC_red_normal.png,Car/RaceC_red_specular.png"
    posx="0" posy="0" posz="0" rotx="-90" roty="0" rotz="0"
    scalex="3" scaley="3" scalez="3">
    <Material diffuse="0.5,0.8,0,1" specularPower="74" specularIntensity="1"/>
    <ComponentList>
        <Component ID="PlayerCar"/>
    </ComponentList>
</GameObject>
```

Except from mainWithRoad.scn

Example of car with different material parameters: Left – high shininess (92) 1 specular intensity, Right – 0 specular intensity

SOUND

See Sound.h and Sound.cpp.

I rely on openAL and alut libraries for playing sound and loading .wav files.

I rely on some lab 9 code for the Sound and extended functionality to make it easier for other objects to retrieve the current state of the sound's source as well as mute the sound.

Like Models, these are loaded by the SceneManager class from that same gamedata file. The file is parsed using tinyxml2 library. Each audio file is loaded into it's own Sound object and those objects are tracked by the ResourceManager.

```
<SoundList>
    <Sound>Slammin.wav</Sound>
    <Sound>carEngine.wav</Sound>
    <Sound>coinPickUp.wav</Sound>
    <Sound>hit1.wav</Sound>
    <Sound>hit2.wav</Sound>
    <Sound>Death.wav</Sound>
    <Sound>end-game-fail.wav</Sound>
</SoundList>
```

The AL_LOOPING flag appeared to not be working so to achieve looping sound effects, I simply had to keep checking if the sound had stopped before I replay it. I loop audio for the player car's engine and for the music.

```
Sound* sound = Game::GetResourceManager()->GetSound("carEngine.wav");
if (sound->GetState() == AL_STOPPED)
    sound->playAudio(AL_NONE);
```

Excerpt from PlayerCar.cpp in the Update(float deltaTime) method.

To mute sound effects, I simply access the sound source and set the volume to 0 (or 1 when un-muting). If a sounds _muted property is true then calls to play the sound are aborted.

```
void Sound::playAudio(ALboolean sndLoop)
{
    if (_muted) return;
    alSourcei(m_OALSource, sndLoop, AL_TRUE);
```

Excerpt from Sound.cpp

The muting of sounds is handled by the Mute(bool state) method which is hooked into an event that is called when the "Mute" input handler's state is changed.

LIGHTS, SHADOW MAPPINGDEFERRED RENDERING AND FOG

See Light.cpp, Light.h and DefRenderer.cpp.

Lights exist in one of 3 states, DIRECTIONAL, POINT and SPOT.

Scenes can only have one directional light at a time and the only thing that matters about this light is it's direction, colour and it's shadow map.

Directional lights are used during the final deferred rendering pass where it's info is used for the appropriate lighting calculation. When a directional light is loaded, it's immediately set to the current scenes main light and during updates, the direction and color vectors are passed to the DefRenderer class.

```
<GameObjectList>
    <GameObject name="Sun" hasRenderer="false" texture="texture.png" model="" shader="Specular"
        posx="0" posy="100" posz="0" rotx="120" roty="0" rotz="0"
        scalex="0.5" scaley="0.5" scalez="2">
        <ComponentList>
            <Component ID="Light">
                <Attributes color="0.5,0.5,1,1" state="directional"/>
            </Component>
        </ComponentList>
    </GameObject>
```

Example of a directional light being set up in the mainWithRoad.scn gamedata file.

Only Directional lights can cast shadows and I currently can't switch directional lights in the scene so if you want a new directional light, you are better off manipulating the game object that has the directional light. I followed a tutorial from LearnOpengl.com and adapted some code from there, I generate a depth texture as a framebuffer in the constructor as well as configure a light orthographic camera. The cameras transform is based on the parent game object the light is attached to. I have render start and end methods that are used to bind the shadow depth texture framebuffer for rendering the scene. To render to the shadow depth texture, I have a render pass where a simple override shader is used to quickly render the scene from the lights point of view.

```
float ShadowCalculation(vec4 fragPosLightSpace)
{
    // perform perspective divide
    vec3 projCoords = fragPosLightSpace.xyz/fragPosLightSpace.w;
    // transform to [0,1] range
    projCoords = projCoords * 0.5 + 0.5;
    // get closest depth value from light's perspective (using [0,1] range fragPosLight as coords)
    float closestDepth = texture(texture3, projCoords.xy).r;
    // get depth of current fragment from light's perspective
    float currentDepth = projCoords.z;
    // check whether current frag pos is in shadow
    float bias = max(0.005 * (1.0 - dot(worldNormal, lightDirection)), 0.0005);

    float shadow = 0.0;

    vec2 texelSize = 1.0 / textureSize(texture3, 0);
    for(int x = -1; x <= 1; ++x)
    {
        for(int y = -1; y <= 1; ++y)
        {
            float pcfDepth = texture(texture3, projCoords.xy + vec2(x, y) * texelSize).r;
            shadow += currentDepth - bias > pcfDepth ? 1.0 : 0.0;
        }
    }
    shadow /= 9.0;
    return shadow;
}
```

Exceprt of one of the shaders used in the geometry stage(roadFS.glsl).

Whether or not a fragment is in shadow is determined in the geometry rendering stage of objects that are taking shadow and the value returned by the calculation is written to the alpha value of the Normal buffer. This value is then read in the final deferred rendering stage and the shadowed fragments are appropriately darkened. ShadowCalculation code is adapted from that shadow mapping learnopengl.com tutorial.

```
float shadowVisibility = texture(texture1, vertexUV).a;
```

…

```
(albedo * diffuseTerm * lightColor * (1 - shadowVisibility))
```

Excerpt from DefRendFS.glsl shader

To keep the shadow map being usable no matter where the scene camera is, the light constantly maintains it's position above the camera. The light's Update(float deltaTime) ensures that the lights position is up to date with both the scene camera .

```
pGameObject->GetTransform()->SetPosition(vec3(newPos.x, newPos.y + 100, newPos.z));
```

There is a limitation with the shadow in that it is currently not set to cover the entire scene as increasing its orthographic view means increasing the texture dimension to ensure that the shadows have an appropriate amount of definition to them so if you move the camera far enough away from objects, their shadows will disappear. Currently, there is also the issue with shadows being calculated in the geometry stage in the shaders that render out the geometry for objects like models, future work would have us move the shadow calculation code to be during one of the lighting passes. This was mainly because at the time of working on them, the position buffer was not being rendered to properly so could not be used to calculate the shadow properly in the final deferred rendering stage.

Point and Spot Lights work very similar. In our deferred renderer, I have a lighting stage where I do a stencil pass that renders the light's model and writes to the stencil buffer, this ensures that fragments where the light doesn't touch are discarded for performance purposes. In the subsequent lighting pass, I use the frame buffers that were written to in the previous geometry rendering stage to calculate our diffuse and specular lighting. I then write to a light buffer which is used in the final deferred renderer stage. To see the entire rendering process and view the various stages, the Game::Render(Camera* cam) method should be looked at (Game.cpp).

The only difference between point and spot lights is the shaders used (PointLightFS.glsl and SpotLightFS.glsl), the models (sphere.fbx and cone.fbx) as well as some differences with the uniforms passed. The default center (vec 3 uniform for shader) of the of the spotlight is incorrect(not at the tip) so I move it by the radius of the model's bounds in the direction of its negative forward vector.

The intensity for the point lights are linear based on distance while the intensity for spot lights degrade to the power of 4 with the distance.

```
float distIntensity = clamp((dist/MaxDistance), 0, 1);

finalColor = (albedo * diffuseTerm * Color.xyz * (1-distIntensity))
+ (spec * specularTerm * Color.xyz * (1-distIntensity));
```

PointLightFS.glsl light intensity calculation

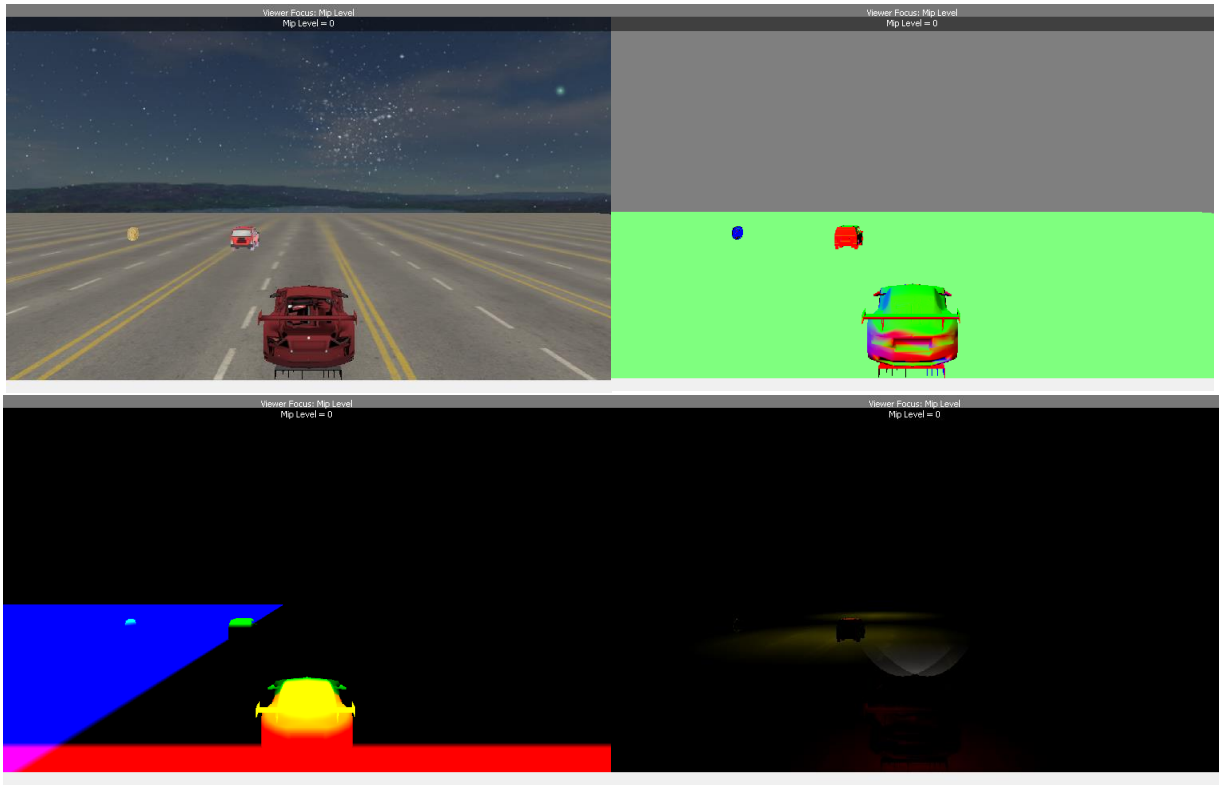
```
float distIntensity = clamp(pow((dist/MaxDistance),4), 0, 1);

vec3 finalColor;
if(FragPos == vec3(0,0,0)) finalColor = vec3(0,0,0);

finalColor = (albedo * diffuseTerm * Color.xyz) * (1-distIntensity);
```

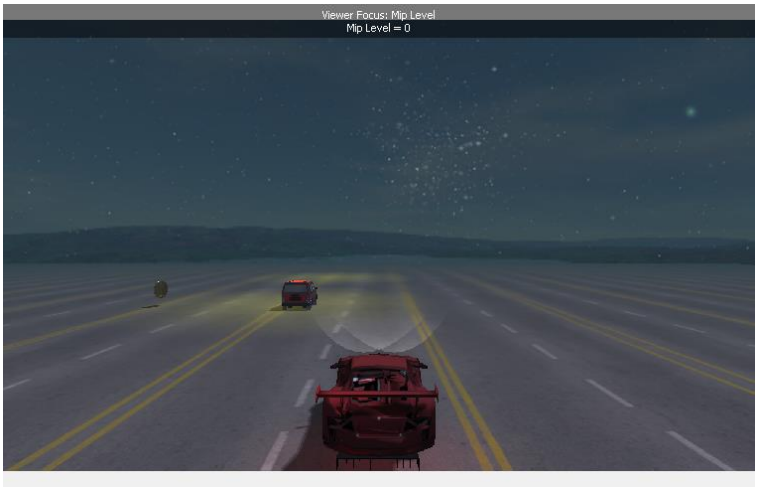
SpotLightFS.glsl light intensity calculation

To prevent lighting calculations from being applied to the skybox, we check if the position buffer value at that fragment is 000, if it is then it must be the skybox so we don't apply lighting there and instead just pass the base color mixed with the fog colour to the finalColor variable.

```
if(FragPos == vec3(0,0,0)) finalColor = mix(albedo, fogColor, 0.5f);
```

To visually illustrate what is going on in the deferred renderer, below is the various buffers and the final result. Some buffers have an alpha channel, so I used these to store some miscellaneous information like specular intensity, objects shininess and shadow visibility. The generation of these buffers can be viewed in the Init() function in DefRenderer.cpp





Left to Right (Color, Normals, Position, Point/Spot Lights and Final – with directional light applied)

The lighting and rendering solutions used here might be considered going above and beyond what was required in the spec with deferred rendering being classified as extension material.

Another element of extension material included was fog, this was easily handled by checking the distance from the camera to a fragment and mixing the color with the fog color using an alpha value calculated by dividing the distance by the fog maxdistance uniform. This was accomplished in the DefRendFS.glsl shader.

```
float dist = distance(cameraPosition, FragPos);
float fIntensity = clamp(dist/fogMaxDistance,0,1);
```

Excerpt taken from DefRendFS.glsl

CAMERAS

See Camera.h, Camera.cpp, Camera Component.h and CameraComponent.cpp

Since I already have a Generic Transform class, our cameras rely on having a parent transform that is used to generate the view matrix. In Scene.h you can see that I only have one camera at a time and to switch camera angles I basically swap the cameras parent transform. This is were camera components come in, these are attached to game objects and when SetSceneTargetCameraToThis is called, the scene camera is essentially hijacked and it's view matrix comes from the gameobject the camera component is attached to.

```
void Camera::Recalculate()
{
    if (_parentTransform != nullptr)
    {
        _parentTransform->GetTransformationMatrix();

        forward = -1.f * _parentTransform->GetForward();
        right = _parentTransform->GetRight();
        up = _parentTransform->GetUp();

        pos = _parentTransform->GetPosition();

        frustrum.UpdateFrustrum(pos, right, up, forward);
        viewMatrix = lookAt(pos, pos + forward, up);

        VP = projMatrix * viewMatrix;
    }
```

Excerpt of camera's recalculation code.

If there is no parent transform assigned to the camera then I recalculate the view matrix using its own position and rotation properties, making use of quaternions to get the rotation matrix (this code hasn't been tested in a while since for our game I rely on parent transforms being set for the camera). Setting the projection matrix simply relies on GLMs own perspective and Ortho functions.

```
quat qPitch = angleAxis(radians(pitch), vec3(1, 0, 0));
quat qYaw = angleAxis(radians(yaw), vec3(0, 1, 0));
quat qRoll = angleAxis(radians(roll), vec3(0, 0, 1));

quat orientation = qPitch * qYaw;
orientation = normalize(orientation);
mat4 currentRotate = mat4_cast(orientation);
```

Excerpt of camera's recalculation code, creating the cameras rotation matrix

In our game, I have 3 cameras that you can switch between, a third person view, an over the bonnet view and a free moving camera. An example of the switching camera logic can be viewed in PlayerCar.cpp in the SwitchCam(bool state) function.

For the free cam, it works a lot like a first person camera in shooters (WASD for movement + mouse for looking around.)

```
//get mouse movement
vec2 deltaPos = Input::GetMouseDelta();
if (aimVertical > 0.2f || aimVertical < -0.2f || aimHorizontal > 0.2f || aimHorizontal < -0.2f)
    deltaPos = vec2(aimHorizontal * aimSensitivity, aimVertical * aimSensitivity);

float deltaPitch = (float)deltaPos.y * mouseSensitivity;
float deltaYaw = (float)deltaPos.x * mouseSensitivity;

quat qPitch = quat(vec3(radians(deltaPitch), 0, 0));
quat qYaw = quat(vec3(0, radians(deltaYaw), 0));

pGameObject->GetTransform()->GetTransformationMatrix();

//combine rotations this way to eliminate unwanted roll
quat total = qPitch * pGameObject->GetTransform()->GetRotation() * qYaw;
pGameObject->GetTransform()->SetRotation(total);
```

When getting the rotation based on mouse movement, you'll notice that I combine the rotations for quat total (local y and x axis) in a deliberate order with the current orientation quaternion of the game object. This was to eliminate any unwanted roll(on the local z) in the game object.

I learned a lot about using quaternions with cameras in this tutorial here:http://in2gpu.com/2016/03/14/opengl-fps-camera-quaternion/

TEXTURES

See Textures.h and Textures.cpp

I have worked with SDL's image loading functionality before, so I used that to load textures, first loading the image as an SDL surface then moving to convert it to a texture in openGL, the surface is immediately freed after this. A few things to note here for texture parameter choices. I set the filtering to GL_Linear since it looks far more favorable to GL_NEAREST as far as textured models are concerned. I also set the textures to be mirrored which allows for some neat panning effects for when I manipulate texture coordinates for the road to make it look like the player is moving.

You'll also notice that I also supported the loading of cube maps which is very important for sky boxes. The implementation is very similar except now I must iterate through binding multiple textures and thankfully glew makes this easy enough to do since the flags for binding the various textures on the cube map are right next to each other

```
#define GL_TEXTURE_CUBE_MAP_POSITIVE_X 0x8515
#define GL_TEXTURE_CUBE_MAP_NEGATIVE_X 0x8516
#define GL_TEXTURE_CUBE_MAP_POSITIVE_Y 0x8517
#define GL_TEXTURE_CUBE_MAP_NEGATIVE_Y 0x8518
#define GL_TEXTURE_CUBE_MAP_POSITIVE_Z 0x8519
#define GL_TEXTURE_CUBE_MAP_NEGATIVE_Z 0x851A
```

Excerpt from glew.h

I followed a tutorial https://learnopengl.com/#!Advanced-OpenGL/Cubemaps to implement cubemaps for my skybox though our implementation differs slightly because of the different loader library and the way we had to integrate skyboxes into our rendering pipeline. If a gameobject has the skybox component, we remove that gameobject from the general gameobjects vector (Scene.cpp) and assign it to the scenes skybox property so we could have more control with when the skybox is rendered (as I must disable depth testing.) This can be seen in the Scene.cpp in the Render(Camera* cam) method. The Skybox component is what handles with setting up the renderer object to render cubemaps for the skybox.

As for passing textures forward to the shader, in the Renderer.cpp Render() function, you can see that the renderer tracks up to 5 textures to pass to the shader, the textures are simply names texture0 up to texture5, this sadly makes it a little hard to track what textures are what in my shaders.

If a renderer object's cubeMap property is true, I simply pass the 2 cubemaps used for the skybox day and night textures. To accomplish the day night cycle, we simply mix the 2 cubemap textures in the skyboxFS.glsl shader based on the time of day. Skybox.cpp has code to take the time of day and calculates a value that can be used as the alpha(0 for day, 1 for night) for the mix while the class TimeDay handles calculating the simulated hour. In the onRender function in skybox.cpp we pass the blendfactor in as a uniform.

COLLISION DETECTION

See Collider.h, Scene.cpp - Update(float deltaTime) function and Enemy.cpp.

For this game, we stick to sphere colliders that check if the distance between the two gameobjects are less than the 2 combined radii of their respective colliders.

All gameobjects can have a collider optionally attached to them, the SphereCol struct only tracks position(set by the gameobject every update) and radius(usually set on cunstruction) and has a function that checks for the collision between another SphereCol. If there is an overlap then the onOverlap(Gameobject * owner) event is fired. This event passes the other gameobject as a parameter so that methods hooked into the event can get access to the object causing the collision.

An example of an onOverlap event being used to dictate game logic can be found in the Enemy.cpp and coin.cpp, these files have definitions of component classes used in the game.

```
void Enemy::OnBegin()
{
    pGameObject->AttachCollider(new SphereCol(4.0f));
    __hook(&SphereCol::onOverlap, pGameObject->GetCollider(), &Enemy::OnOverlap);
```

…

```
void Enemy::OnOverlap(GameObject * other)
{
    PlayerCar* pc = dynamic_cast<PlayerCar*>(other->GetComponent("PlayerCar"));
    if (pc != NULL)
    {
        Game::GetResourceManager()->GetSound("hit1.wav")->playAudio(AL_NONE);
        pGameObject->Destroy();
        pc->AddHealth(-1);
    }
}
```

Excerpt from Enemy.cpp

We don't have any broad phase and instead simply iterate through every gameobject in the scene and do a collision check with every other game object. The collision check occurs right after we do the gameObjects updates in the Scene.cpp Update(float deltaTime) function.

```
//Sphere Collisions
for (int i = 0; i != gameObjects.size(); i++)
{
    if (gameObjects.at(i)->GetCollider() == NULL) continue;

    for (int j = 0; j != gameObjects.size(); j++)
    {
        if (i == j || gameObjects.at(j)->GetCollider() == NULL) continue;

        if (gameObjects.at(i)->GetCollider()->isOverlap(gameObjects.at(j)->GetCollider()))
        {
            gameObjects.at(i)->GetCollider()->onOverlap(gameObjects.at(j));
        }
    }
}
```

INPUT (KEYBOARD TOGGLES)

See Input.h, Input.cpp, inputAction.h and Game.cpp

Our Input class is mainly filled static functions and properties which makes it easy any object to access the input manager to retrieve states of keys and gamepad inputs.

In the Game.cpp Update method, we first poll events using SDL, taking key down, key up and gamepad events then updating the states of those keys and gamepad buttons in our input manager.

The 'raw' states of these keys and buttons can be accessed pretty easily by other objects. However, after watching this steam controller presentation (https://youtu.be/7I4SiAiKqqk?t=10m51s) I decided to implement abstracted inputs so that game logic and raw inputs are kept separate. To do this, I have inputActions and inputAxis, these structs store what keys and gamepad inputs they are tied to and they have a state (Boolean for input actions, float for inputAxis). In the methods Input::UpdateInputActions() and Input::UpdateInputAxis(), we iterate through all currently loaded inputActions/Axis and update their states by checking the raw inputs they are tied to and after that is done we fire an event tied to the inputAction/Axis (for input actions we only fire an event if a change in state was registered).

Hooking to events works much the same way it did for colliders and examples can be easily seen in PlayerCar.cpp and cameraBehaviour.cpp.

```
InputAxis* iaH = Input::GetInputAxisState("HorizontalCar");
if (iaH)
    __hook(&InputAxis::InputAxisChange, iaH, &PlayerCar::strafe);

InputAction* iaS = Input::GetInputActionState("SwitchCam");
if (iaS)
    __hook(&InputAction::InputActionChange, iaS, &PlayerCar::switchCam);

InputAction* iaF = Input::GetInputActionState("Debug");
if (iaF)
    __hook(&InputAction::InputActionChange, iaF, &PlayerCar::FreeCam);
```

...

```
void PlayerCar::strafe(float state)
{
    if(!debug)
    strafeAccel = state * Game::GetGlobalDeltaTime() * 1000;
}
```

...

```
void PlayerCar::switchCam(bool state)
{
    if (state && pGameObject->GetActive() && !debug)
    {
        firstPerson = !firstPerson;
        if (firstPerson)
        {
            _FirstPersonCam->SetSceneTargetCameraToThis();
        }
        else _ThirdPersonCam->SetSceneTargetCameraToThis();
    }
}
```

Excerpts taken from PlayerCar.cpp

The benefit this approach is that component Update() methods are kept cleaner as we don't bother having them check the input manager every update.

You will notice that at no point in the code are the inputActions/Axis buttons hard coded as they in fact loaded via parsing an XML file, this makes for easy rebinding of keys without having to recompile code.

Once again, tinyxml2 library is used and the input loader function is called Input::LoadInput(). It was easy to load most key presses since we only need to retrieve a char which can be easily translated to the correct SDL_KEYCODE but as for gamepad buttons and non-alphanumeric keys, we rely on hard coded lookup tables that can be viewed in CommonXML.cpp.

```
const map<string, Uint8> gamepadStrings =
{
    { "gamepad_a", SDL_CONTROLLER_BUTTON_A },
    { "gamepad_b", SDL_CONTROLLER_BUTTON_B },
    { "gamepad_x", SDL_CONTROLLER_BUTTON_X },
    { "gamepad_y", SDL_CONTROLLER_BUTTON_Y },
    { "gamepad_lb", SDL_CONTROLLER_BUTTON_LEFTSHOULDER },
    { "gamepad_rb", SDL_CONTROLLER_BUTTON_RIGHTSHOULDER },
    { "gamepad_back", SDL_CONTROLLER_BUTTON_BACK },
    { "gamepad_start", SDL_CONTROLLER_BUTTON_START },
    { "gamepad_guide", SDL_CONTROLLER_BUTTON_GUIDE },
    { "gamepad_ls", SDL_CONTROLLER_BUTTON_LEFTSTICK },
    { "gamepad_rs", SDL_CONTROLLER_BUTTON_RIGHTSTICK },
    { "gamepad_up", SDL_CONTROLLER_BUTTON_DPAD_UP },
    { "gamepad_down", SDL_CONTROLLER_BUTTON_DPAD_DOWN },
    { "gamepad_left", SDL_CONTROLLER_BUTTON_DPAD_LEFT },
    { "gamepad_right", SDL_CONTROLLER_BUTTON_DPAD_RIGHT },
    { "gamepad_leftStickY", SDL_CONTROLLER_AXIS_LEFTY },
    { "gamepad_leftStickX", SDL_CONTROLLER_AXIS_LEFTX },
    { "gamepad_RightStickY", SDL_CONTROLLER_AXIS_RIGHTY },
    { "gamepad_RightStickX", SDL_CONTROLLER_AXIS_RIGHTX },
    { "gamepad_LT", SDL_CONTROLLER_AXIS_TRIGGERLEFT },
    { "gamepad_RT", SDL_CONTROLLER_AXIS_TRIGGERRIGHT }
};

const map<string, SDL_Keycode> specialKeyStrings =
{
    { "left", SDLK_LEFT },
    { "right", SDLK_RIGHT },
    { "up", SDLK_UP },
    { "down", SDLK_DOWN },
    { "l_shift", SDLK_LSHIFT },
    { "r_shift", SDLK_RSHIFT },
    { "tab", SDLK_TAB },
    { "l_ctrl", SDLK_LCTRL },
    { "r_ctrl", SDLK_RCTRL },
    { "space", SDLK_SPACE },
    { "l_alt", SDLK_LALT },
    { "l_alt", SDLK_RALT },
};
```

The results were quite pleasing and now it's easy for me as a programmer to check the inputs I am using for the game and change them on the fly without recompiling code.

```
<Input>
    <Actions>
        <IA title = "Debug" keyboard = "tab,f" gamepad = "gamepad_y"/>
        <IA title = "SpeedBoost" keyboard = "l_shift" gamepad = "gamepad_a"/>
        <IA title = "SpeedBoostToggle" keyboard = "x" gamepad = "gamepad_ls"/>

        <IA title = "SwitchCam" keyboard = "c" gamepad = "gamepad_lb"/>
        <IA title = "Mute" keyboard = "x" gamepad = "gamepad_b"/>
        <IA title = "Restart" keyboard = "r" gamepad = "gamepad_back"/>
    </Actions>
    <Axis>
        <IA title = "Vertical" keyboardPositive = "w" keyboardNegative = "s" ga
        <IA title = "Horizontal" keyboardPositive = "d" keyboardNegative = "a"

        <IA title = "HorizontalCar" keyboardPositive = "right" keyboardNegative
        <IA title = "AimVertical" keyboardPositive = "" keyboardNegative = "" g
        <IA title = "AimHorizontal" keyboardPositive = "" keyboardNegative = ""
        <IA title = "Hover" keyboardPositive = "e" keyboardNegative = "q" gamep
    </Axis>
</Input>
```

Excerpt from input.xml (assets/gamedata folder)

As mentioned before, gamepads do work for the game with the only concession being that there is currently no vibration functionality.

KEYBOARD TOGGLES

In the coursework brief, we were asked to add keyboard toggles for muting sounds as well as switching camera perspective. These are implemented to specification with “Debug” and “SwitchCam” inputs switching cameras (an excerpt of the switch camera logic was included above, for more info on how camera switching is done, please got to the Cameras page.) To mute sounds, the “Mute” input is pressed, the logic for this implementation can be viewed in Sounds.cpp.

MOVING MODELS WITH INPUT

See PlayerCar.cpp.

In the game, the player has control over the main player car model as they move to avoid traffic. We hook a strafe method to the inputAction’s (“HorizontalCar”) InputActionChange event and when that is fired off the local left and right acceleration updated. In the Update() method we take this acceleration and add it to the acceleration vector of the car, multiplying it with the “right” direction vector. After adding the acceleration vector to the velocity vector, we can use the AddPosition() method of the gameObjects transform to move the car in the appropriate direction with a subtle acceleration and deceleration to make it feel a little more natural. To handle deceleration, we interpolate velocity vector to 0 (see Common.cpp). To prevent the car from leaving the level, we simply clamp the cars position on the z axis passing in hard coded values that represent the left and right bounds of the level.

The Enemy cars are much simpler as they just move in a single direction at a constant speed. The EnemyRC component moves it’s car along a sine wave over time to add a layer of challenge for the player to deal with.

See EnemySpawner.cpp

The EnemySpawner Class is a component attached to a gameobject in the level that handles spawning of the enemy and coin gameObjects. I relied on the simple Rand() function to affect the likelihood of the EnemyRC car spawning, Though this was not appropriate when randomising the positions or the base Enemy car as we spawn multiples of those across 8 lanes, so to avoid reusing the same spawn position, we create an array of unique integers and shuffle them before stepping through them, ensuring that there are no overlapping enemy cars. The randomisation of unique numbers code was based off of code in this forum http://www.cplusplus.com/forum/beginner/91557/.

FONTS

See Font.h and Font.cpp.

We rely on SDL ttf font loader to convert the true type font file to a SDL surface then to a texture that is usable in OpenGL.

In the font's render method, we build up the vertices and indices for the planes, assigning the correct texture coordinates to ensure that they render with the correct character from the font texture. Render() doesn't actually draw anything just yet, it can be called at anytime throughout the update and render phases in game objects and their components (as it is easy for them to retrieve the font from the global resource manager.)

Flush() is the last thing called in our render pipeline (can be viewed in render() in game.cpp) as we actually render out our text and proceed to clear the vertices and indices for the next update.

The implementation isn't perfect as we never attempt to maintain the proper aspect ration of a character as the planes dimensions are determined by the SDL_Rect passed as a parameter to render() calls which can lead to text appearing stretched if not called properly.

REFERENCES

ART ASSETS

- Car SUV: https://www.cgtrader.com/free-3d-models/car/suv/low-poly-fire-car-suv
- Race Car & Regular Car: https://assetstore.unity.com/packages/3d/vehicles/land/lowpoly-sports-car-2-in-1-52995
- Sky box Textures: http://www.custommapmakers.org/skyboxes.php
- Regebsburg font: http://www.1001fonts.com/regensburg-font.html
- Resource used to create normal and spec maps: http://cpetry.github.io/NormalMap-Online/
- Coin: https://free3d.com/3d-model/coin-4532.html

SOUND

- Music: http://www.purple-planet.com/dance/4583971242
- Crash: https://freesound.org/people/FxKid2/sounds/367624/
- Car hit: https://freesound.org/people/qubodup/sounds/151624/
- Car hit 2: https://freesound.org/people/PaulMorek/sounds/196724/
- Coin pick up: https://freesound.org/people/GameAudio/sounds/220173/
- End Game Fail: https://freesound.org/people/davidbain/sounds/135831/

CODE RESOURCES:

- Random unique number generator: http://www.cplusplus.com/forum/beginner/91557/
- Shadow mapping: https://learnopengl.com/#!Advanced-Lighting/Shadows/Shadow-Mapping
    - Shadow Calculation code in shader's like roadFS.glsl is adapted from code in this resource
- http://www.euclideanspace.com/maths/geometry/rotations/conversions/quaternionToEuler/
    - Code used in Transform.cpp to convert quaternion to euler
- To learn more about quaternions: http://www.opengl-tutorial.org/intermediate-tutorials/tutorial-17-quaternions/
- To do a first person camera with quaternions: //in2gpu.com/2016/03/14/opengl-fps-camera-quaternion/
    - Code from this resource was adapted for recalculate in camera.cpp and for cameraBehaviour.cpp.
    - Code was also adapted to create the shadow texture and set up the light, can be seen in Light.cpp's constructor.
- Tinyxml2 .cpp and Tinyxml2.h was included directly in the project following the instructions for including the library by the documentation online. To learn about tinyxml2: http://leethomason.github.io/tinyxml2/
- To learn about creating framebuffers: https://learnopengl.com/#!Advanced-OpenGL/Framebuffers
- To learn more about deferred rendering: https://learnopengl.com/#!Advanced-Lighting/Deferred-Shading

- Learn more about SDL 2 like gamepad related stuff: https://www.youtube.com/watch?v=MeMPCSqQ-34
- To learn about cubemap creation: https://learnopengl.com/#!Advanced-OpenGL/Cubemaps