Portfolio 2 – 3DGE

Pramod Nepal

Contents

Lab 08 (Basic Illumination)

The objective of this lab was to add light into the scene. This lab also demonstrates the construction of a room. In the lab I also added a basic animation to the light source, so that it would demonstrate a back forth behavior just like patrolling cubes and objects in previous projects.

The light source is defined in LocalLightSource class. The light intensity is passed to the GPU in OGLGraphicsSystem's render method.

```
float lightIntensity = this->gameWorld->getLocalLightSource()->getIntensity();
this->oglShaderManager->sendValueToGPU(
              this->activeShaderProgram,
              "lightIntensity",
              lightIntensity);
```

In the GameEngine class's setupGame behavior, initial position and intensity is set.

```
LocalLightSource *localLightSource = (LocalLightSource *)
              this->graphics->getGameWorld()->getLocalLightSource();

       localLightSource->setPosition(glm::vec3(-8, 9, -8));
       localLightSource->setIntensity(0.2f);
       localLightSource->setBehavior(new LightBackForthBehavior(15));
```

As can be seen from above code, the behavior for the light source back forth behavior is created in LightBackForthBehavior class. The LocalLightSource object is passed to the LightBackForthBehavior class's update method. In this method position and intensity is changed gradually using a small delta value. The intensity is set in the range 0 to maxIntensity. Similarly the distance moved is also calculated between 0 and maxDistance. The maxIntensity is set to 1 and the maxDistance is passed in the constructor. As seen from above code maxDistance is set to 15. The light source moves forward and backward, therefore only two state variables are used for changing the movement. Following code snippet shows the update method.

```
void LightBackForthBehavior::update(LocalLightSource *lightSource, float elapsedSeconds)
{
       LocalLightSource* obj = (LocalLightSource*)lightSource;
       float delta = 10.0f * elapsedSeconds;
       this->distanceMoved += delta;
       this->intensity += delta * 0.04;

       glm::vec3 position = lightSource->getPostion();
       glm::vec3 newPosition;

       lightSource->setIntensity(this->intensity);

       // Change intensity
       if (this->intensity >= this->maxIntensity) {
              this->intensity = this->intensity - this->maxIntensity;
       }

       switch (this->state) {
       case MOVING_BACKWARD:
              if (this->distanceMoved >= this->maxDistance) {
                     this->state = MOVING_FORWARD;
```

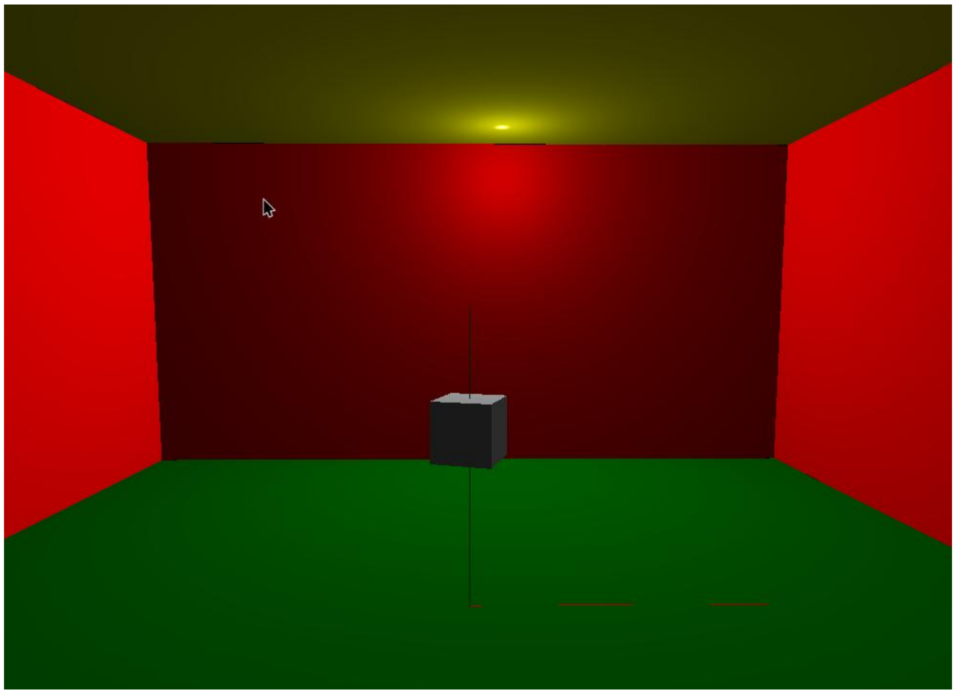
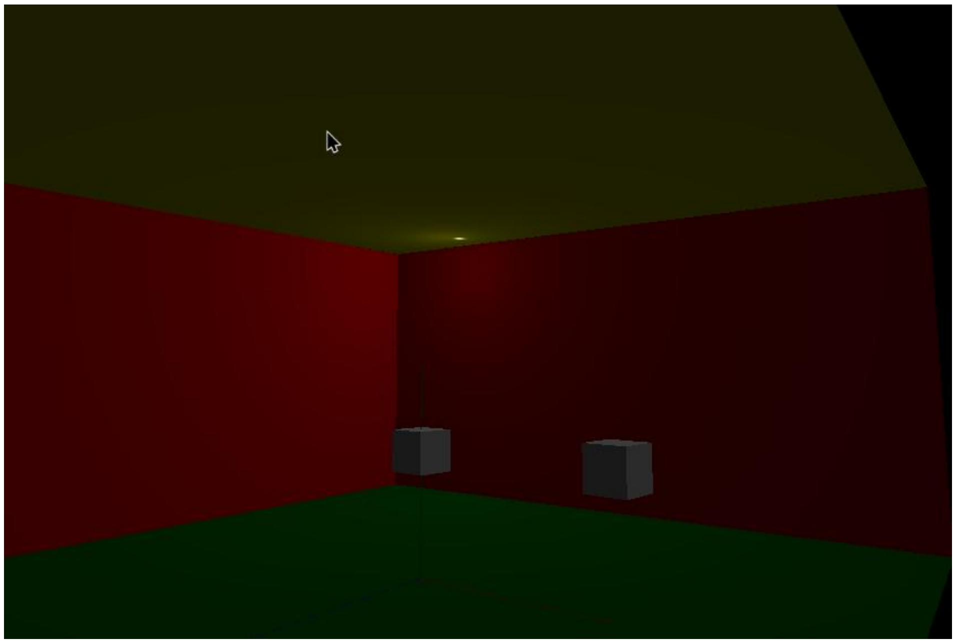
```
					delta = this->distanceMoved - this->maxDistance;
					this->distanceMoved = 0;
				}

				newPosition = position;
				newPosition.x -= delta;
				lightSource->setPosition(newPosition);

				break;
			case MOVING_FORWARD:
				if (this->distanceMoved >= this->maxDistance) {
					this->state = MOVING_BACKWARD;
					delta = this->distanceMoved - this->maxDistance;
					this->distanceMoved = 0;
				}

				newPosition = position;
				newPosition.x += delta;
				lightSource->setPosition(newPosition);
				break;
		}
}
```

In addition to adding light and its behavior, two more cubes are added to the scene with rotate (in y-axis) and four point patrol behavior (move in a square pattern).

The walls are constructed using ObjectGenerator’s generateFlatSurface method. The wall creation code is added in StockObjectLoader’s loadObjects method.

```
	object = new OGL3DObject("Right Wall");
	object->setIndexedArrayType();
	arr = ObjectGenerator::generateFlatSurface(10, 10, 10, 20, { 1.0f, 0.0f, 0.0f,
1.0f });
	object->setVertexData(arr.vertexData);
	object->setIndexData(arr.indexData);
	vboObject = OGLObject::createVBOObject(
		"triangles", arr.vertexData, arr.indexData, GL_TRIANGLES);
	object->addVBOObject(vboObject);
	object->referenceFrame.move(glm::vec3(1, 0, 0), 10);
	object->referenceFrame.move(glm::vec3(0, 1, 0), 5);
	object->referenceFrame.rotateZ(90);
	gameObjectManager->addObject("Right Wall", object);
```

The final output can be seen in following screenshots. As can be seen from the outputs the intensity and position of light changes. There is a cube that is rotating along the y-axis in the center and there is also another cube that has the patrolling behavior.

Lab 09 (Hierarchical Models)

The objective of this lab was to learn to create objects in reference to other objects. I created a hierarchical model using stack so that objects are positioned in reference to the objects that make more relevance in terms of being its base object. E.g. while creating an arm that has individual graphics objects such as shoulder, bicep, forearm and hand, it makes more sense to use shoulder as base and reference next object that follows. The benefit of having a hierarchical model is that the object can be referenced as a whole rather than parts that compose it. In the lab I created an arm and a sword. I then attach the sword to the arm and add chopping behavior to our arm. Since we add objects in hierarchical model in both Arm and Sword, when the arm moves it moves the whole sword object that is composed of individual parts like hilt, cross guard and blade. All of these individual objects are composed using Cuboid. The Arm class's render method shows the hierarchical model.

```
void Arm::render()
{
	this->shoulder->referenceFrame = this->referenceFrame;
	this->frameStack.setBaseFrame(this->shoulder->referenceFrame);
	this->shoulder->render();

	this->frameStack.push();
	{
		this->frameStack.translate(0, -0.775f, 0);
		this->frameStack.rotateX(90.0f);
		this->bicep->referenceFrame = this->frameStack.top();
		this->bicep->render();

		this->frameStack.push();
		{
			this->frameStack.translate(0, 0, 0.625f);
			this->frameStack.rotateX(angleX);
			this->frameStack.translate(0, 0, 0.625f);
			this->forearm->referenceFrame = this->frameStack.top();
			this->forearm->render();
			this->axis->render(this->frameStack.top());

			this->frameStack.push();                  {

				this->frameStack.translate(0, 0, 0.725f);
				this->hand->referenceFrame = this->frameStack.top();
				this->hand->render();

				this->frameStack.push();
				{
					this->frameStack.rotateY(90.0f);
					this->sword->referenceFrame = this->frameStack.top();
					this->sword->render();
				}
				this->frameStack.pop();
			}
			this->frameStack.pop();
		}
		this->frameStack.pop();
	}
	this->frameStack.pop();
}
```

As can be seen from above code shoulder is the base object. The bicep, forearm, hand and sword are then rendered in a hierarchical order. Just like the above code Sword class's render method renders its individual parts in a hierarchical order.

The sword object is added to the arm in StockObjectLoader's loadObjects method.

```
Sword *katana = new Sword("Katana");
	gameObjectManager->addObject("Katana", katana);

	Arm *arm = new Arm("Left Arm");
	arm->addSword(katana);
	gameObjectManager->addObject("Left Arm", arm);
```

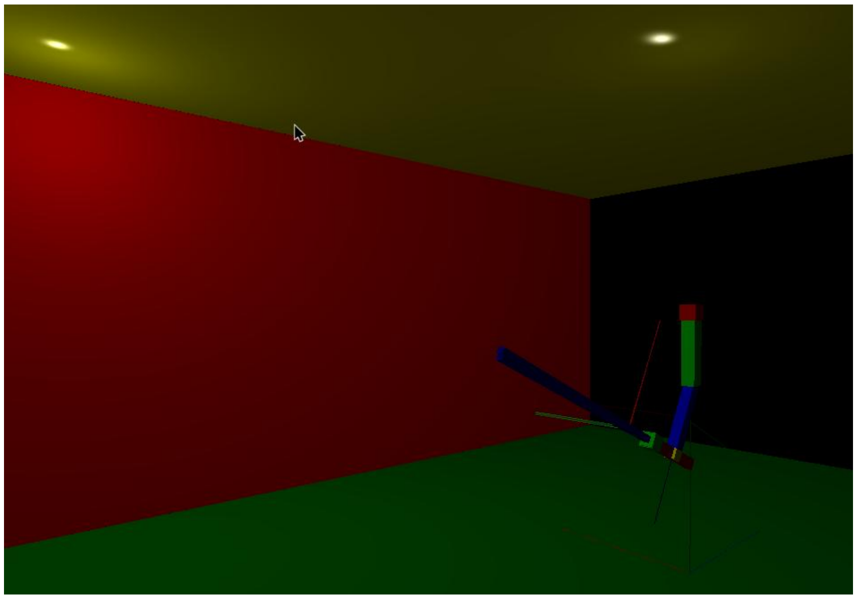
The chopping or swing action is achieved by moving the arm in up and down movement. This is done in Arm's update method. The objective of this method is to create the movement action along the X-axis. It changes the angleX variable. This variable is used to move the forearm in the render method, as can be seen from above code

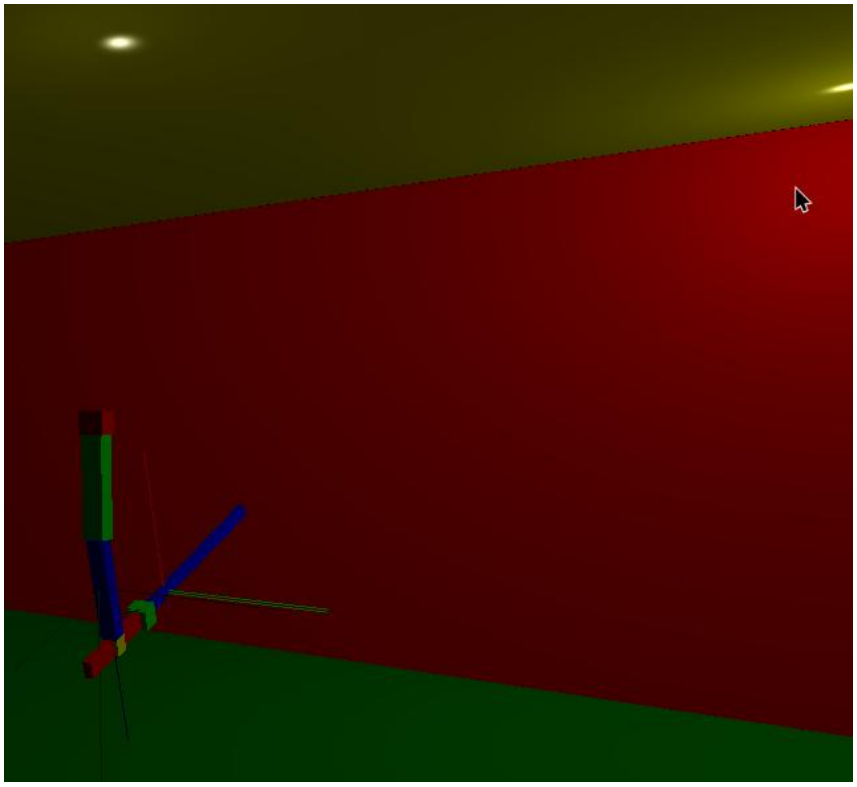
```
void Arm::update(float elapsedSeconds)
{
	OGL3DCompositeObject::update(elapsedSeconds);

	switch (this->state) {
		case UP_MOVEMENT: {
			this->angleX += (30 * elapsedSeconds);
			if (this->angleX > 30) {
				this->state = DOWN_MOVEMENT;
			}
			break;
		}
		case DOWN_MOVEMENT: {
			this->angleX -= (30 * elapsedSeconds);
			if (this->angleX < 0) {
				this->state = UP_MOVEMENT;
			}
			break;
		}

	}
}
```

Multiple light sources are added by adding them to GameWorld's LightSource vector called localLights. Two light sources are added in TheGame's setup method.

```
	LightSource *light = new LightSource();
	light->setPosition(8.0f, 9.0f, -8.0f);
	light->setIntensity(0.4f);
	graphics->getGameWorld()->localLights.push_back(light);

	light = new LightSource();
	light->setPosition(0.0f, 9.0f, 0.0f);
	light->setIntensity(0.1f);
	graphics->getGameWorld()->localLights.push_back(light);
```

The final output can be seen in following screenshots. The screenshot shows two light sources in the room. It also shows the sword attached to the arm (hand) and swinging.





Lab 10 (Basic Texture Mapping)

The objective of this lab was to learn to create objects with textures.

I start the lab by creating a 2D plane in a class called Plane. The Plane class uses similar method to create objects as I did for earlier labs. However, the Plane's generate method uses 13 values for each vertex instead of previous 11. The 13 values are divided as 4 for the position, 4 for the colors, 3 for the normal and 2 for texture. Following code snippet shows the texture addition code in Plane's generate method.

```
	// Texture Coordinates
	// A
	vertexData[i++] = 0.0f;
	vertexData[i++] = 1.0f;
	// B
	vertexData[i++] = 0.0f;
	vertexData[i++] = 0.0f;
	// C
	vertexData[i++] = 1.0f;
	vertexData[i++] = 0.0f;
	// A
	vertexData[i++] = 0.0f;
	vertexData[i++] = 1.0f;
	// C
	vertexData[i++] = 1.0f;
	vertexData[i++] = 0.0f;
	// D
	vertexData[i++] = 1.0f;
	vertexData[i++] = 1.0f;
```

As seen from above code, the vertices use different corners of the texture. The coordinate A uses the left-top, B uses left-bottom, C uses right-bottom and D uses right-top corners of the texture data.

A 2D texture is defined in CustomTexture class. CustomTexture's create method creates a new texture. It uses color instead of image file to define the texture.

```
void CustomTexture::create()
{
	glGenTextures(1, &this->id);
	this->select();
	float data[] = {
		1.0f, 0.0f, 0.0f,  0.0f, 0.0f, 0.01f,  0.0f, 0.0f, 0.01f,  1.0f, 0.0f, 
0.0f,
		1.0f, 0.0f, 0.0f,  1.0f, 0.0f, 0.0f,   1.0f, 0.0f, 0.0f,   1.0f, 0.0f, 
0.0f,
		1.0f, 0.0f, 0.0f,  0.0f, 0.0f, 0.01f,  0.0f, 0.0f, 0.01f,  1.0f, 0.0f, 
0.0f,
		0.0f, 0.0f, 0.01f, 1.0f, 0.0f, 0.0f,   1.0f, 0.0f, 0.0f,   0.0f, 0.0f, 
0.01f
	};

	glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, 4, 4, 0, GL_RGB, GL_FLOAT, data);
	glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
	glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
}
```

OGL2DTexture class defines texture using 2D image bitmap Texture object passed to its constructor. The Plane object is given this texture in the TheGame's setup method.

```
shader = (OGLShaderProgram*)shaderMgr->getShader("TextureShader");
       bmpLoader.setFilePath("wood.bmp");
       OGL2DTexture* tex = new OGL2DTexture(bmpLoader.load());
       tex->setTexelFormat(GL_BGR);
       tex->setTypeOfData(GL_UNSIGNED_BYTE);
       tex->create();

       Plane *plane = new Plane(3, 3);
       plane->setTexture(tex);
       plane->referenceFrame.setPosition(0, 5, 0);
       plane->setShaderProgram(shader->getHandle());
       graphics->getGameWorld()->getObjectManager()->addObject("Plane", plane);
```

Using code from Cuboid and Plane classes I create another class called TexturedCuboid. This class's generateTexturedCuboid method creates a cuboid object using similar code as a Cuboid class. However it adds the texture values to each of the vertices.

```
       for (int k = 0; k < 6; k++) {
              // Texture Coordinates
              // A
              vertexData[i++] = 0.0f;
              vertexData[i++] = 1.0f;
              // B
              vertexData[i++] = 0.0f;
              vertexData[i++] = 0.0f;
              // C
              vertexData[i++] = 1.0f;
              vertexData[i++] = 0.0f;
              // D
              vertexData[i++] = 1.0f;
              vertexData[i++] = 1.0f;
       }
```

TexturedCuboid's generate method uses the ElementArray object returned from above method and creates a textured cuboid by calling OGLObject's createTextureVBOObject method.

```
void TexturedCuboid::generate()
{
       this->setIndexedArrayType();
       ElementArray arr = this->generateTexturedCuboid(
              this->width, this->depth, this->height, this->faceColor);
       this->setVertexData(arr.vertexData);
       this->setIndexData(arr.indexData);


       VBOObject* vboObject = OGLObject::createTextureVBOObject(
              "triangles", arr.vertexData, arr.indexData, GL_TRIANGLES);
       this->addVBOObject(vboObject);
}
```

In this lab I also experimented with transparency. The transparency code is changed in the FragmentShaderTexture.glsl file.

```
if(texFragColor.r == 0 && texFragColor.g == 0 && (texFragColor.b <= .02 && texFragColor.b
>= 0.01)){
	discard;
}else{
	color = totalFragColor
		+ fragGlobalColor
		+ (materialSpecular * globalPhongTerm * fragGlobalLightIntensity)
		+ totalSpecular
		+ (texFragColor * materialAmbientIntensity);
}
```

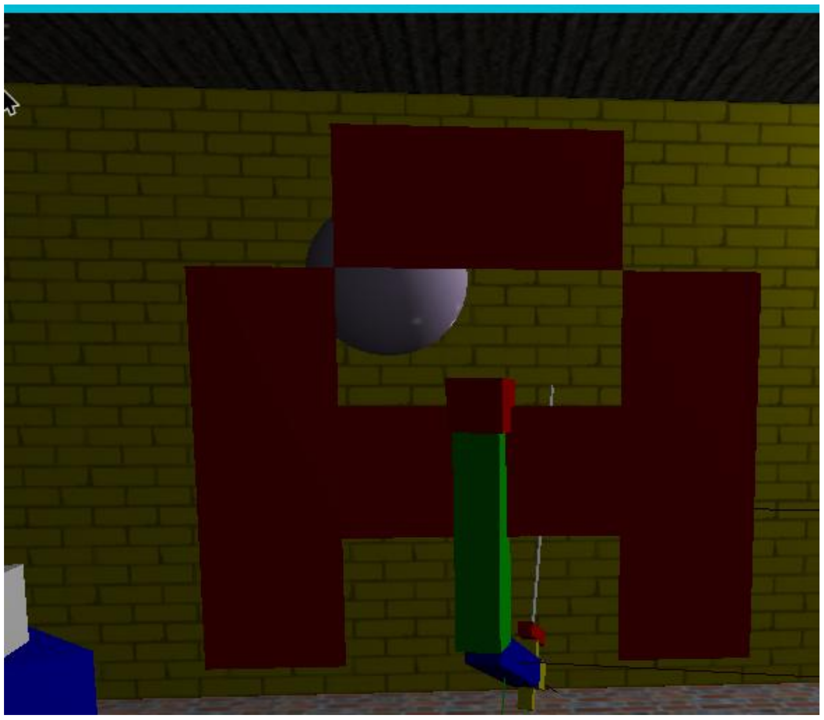
The shader code checks whether the color values is 0 for red, 0 for green and between 0.01 and 0.02. If the color values fall within this range it is discarded else it is drawn.

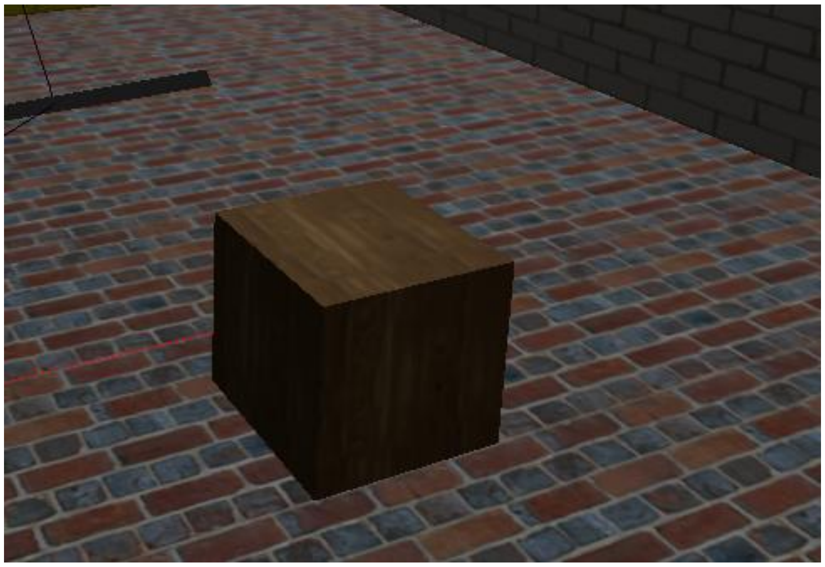
In the final part of the lab using the TexturedCuboid I created a Chest object. A Chest object is constructed using two cuboids (one as the base and another as the lid). I also add a behavior to the lid, so that it would animate open and close behavior. Chest's update method transitions between MOVE_UP and MOVE_DOWN states and update the rotateDegrees variable and Chest's render method uses this variable to rotate the lid. Following code snippet shows this behavior.

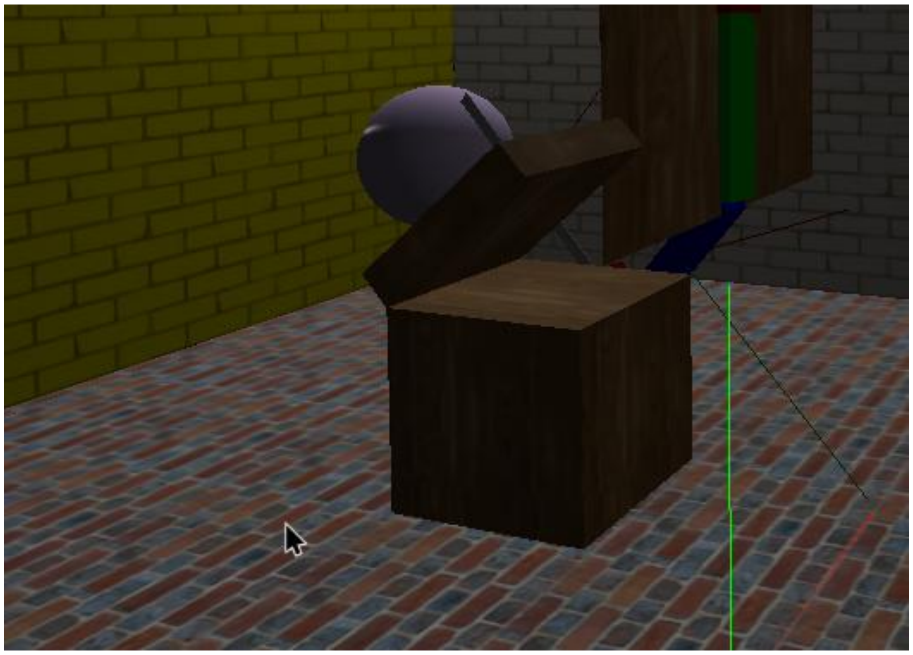
```
void Chest::render()
{
	this->base->referenceFrame = this->referenceFrame;
	this->frameStack.setBaseFrame(this->base->referenceFrame);
	this->base->render();
	this->frameStack.push();
	{
		this->frameStack.translate(0.0, 0.5f, -0.5f);
		this->frameStack.rotateX(-this->rotateDegrees);
		this->frameStack.translate(0, 0.1, 0.5);

		this->lid->referenceFrame = this->frameStack.top();
		this->lid->render();
	}
	this->frameStack.pop();
}

void Chest::update(float elapsedSeconds)
{
	OGL3DCompositeObject::update(elapsedSeconds);

	switch (this->state) {
	case MOVE_UP:
		this->rotateDegrees += (90.0f * elapsedSeconds);
		if (this->rotateDegrees > 75.0f) {
			this->state = MOVE_DOWN;
		}
		break;
	case MOVE_DOWN:
		this->rotateDegrees -= (90.0f * elapsedSeconds);
		if (this->rotateDegrees < 0.0f) {
			this->state = MOVE_UP;
```

```
                    }
                    break;
          }
}
```

Following screenshots show both the transparent plane, textured cuboid and the textured chest object.







Lab 11 (Your Object)

The objective of this lab was to learn to create objects with textures.

A new way of creating vertex data is added to this lab. ObjectGenerator class has maps for positions, colors, normals and texture coordinates. There are two vectors for triangles and lines.

```
        map<unsigned int, Position> positions;
        map<unsigned int, Color> colors;
        map<unsigned int, Normal> normals;
        map<unsigned int, Tex> textureCoords;
        vector<Triangle> triangles;
        vector<Line> lines;
```

A class can use these structures to define the points, colors, normals and textures separately. MyObject's generate method shows its use.

```
void MyObject::generate()
{
        ObjectGenerator gen;
        gen.clear();

        // Positions{x,y,z,w}
        gen.positions[0] = { 0, 1, 0, 1 };
        gen.positions[1] = { -1, -1, 0, 1 };
        gen.positions[2] = { 1, -1, 0, 1 };

        // Colors{r, g, b, a}
        gen.colors[0] = { 1, 0, 0, 1 };
        gen.colors[1] = { 1, 0, 0, 1 };
        gen.colors[2] = { 1, 0, 0, 1 };

        // Normals {x, y, z}
        gen.normals[0] = { 0, 0, 1 };
        gen.normals[1] = { 0, 0, 1 };
        gen.normals[2] = { 0, 0, 1 };


        // Indexes
        gen.triangles.push_back({ 0, 1, 2 });

        float *vertexData = gen.generateVertexData();
        short *indexData = gen.generateIndexData();
        this->createElementArrayPCN("Triangles", vertexData, indexData, GL_TRIANGLES);
}
```

As seen from above code positions, colors and normals are defined separately. The vertices are pushed back to the triangles and MyObject's appropriate object construction method is called. In this case createElementArrayPCN method is used. This method creates the object using position, color and normal data. Here I created a triangle that uses three vertices. I add this object to the application in TheGame's setup method.

```
        OGLShaderProgram *shader = (OGLShaderProgram *)shaderMgr-
>getShader("ShaderProgramIllumination");
```

```
        object = new MyObject();
        object->referenceFrame.setPosition(0, 5, 0);
        object->setShaderProgram(shader->getHandle());
        objectMgr->addObject(object->getName(), object);
        object->setVisibility(true);
```

In the next part of the lab, I added a texture to the triangle. To create a textured triangle all it takes is to add the texuture data to above code and call the appropriate object creation method. I added texture coordinates to MyObject's generate method as shown in following code snippet.

```
// Texture Coordinates {s, t}
        gen.textureCoords[0] = { 0.5f, 1 };
        gen.textureCoords[1] = { 0, 0 };
        gen.textureCoords[2] = { 1, 0 };
```

Similarly, I use createElementArrayPCNT (t for texture) instead of createElementArrayPCN method. In TheGame class's setup method, I changed the shader to TextureShader, added a new OGL2DTexture object and added it to MyObject's object.

```
OGLShaderProgram *shader = (OGLShaderProgram *)shaderMgr->getShader("TextureShader");
texture = (OGL2DTexture *)texMgr->getTexture("WallTex");
object = new MyObject();
object->setTexture(texture);
```

Next task in the lab was to create an object that is used in the Final team project. I selected Pawn as the object to create from the final project in this lab. For the purpose of this lab the Pawn class uses three TexturedCuboid as head, body and feet.

```
TexturedCuboid *head;
TexturedCuboid *body;
TexturedCuboid *feet;
```

Pawn's render method renders the cuboid stacked one over another giving it the pawn's shape. It uses the referenceFrame stack discussed in previous labs.

```
void Pawn::render()
{
        this->head->referenceFrame = this->referenceFrame;
        this->head->referenceFrame.translate(0, 1, 0);
        this->frameStack.setBaseFrame(this->head->referenceFrame);
        this->head->render();
        this->frameStack.push();
        {
                this->frameStack.translate(0, -0.5, 0);
                this->frameStack.rotateX(180);
                this->body->render(this->frameStack.top());
                this->frameStack.push();
                {
                        this->frameStack.translate(0, 1, 0);
                        this->feet->render(this->frameStack.top());
                }
                this->frameStack.pop();
        }
        this->frameStack.pop();
}
```

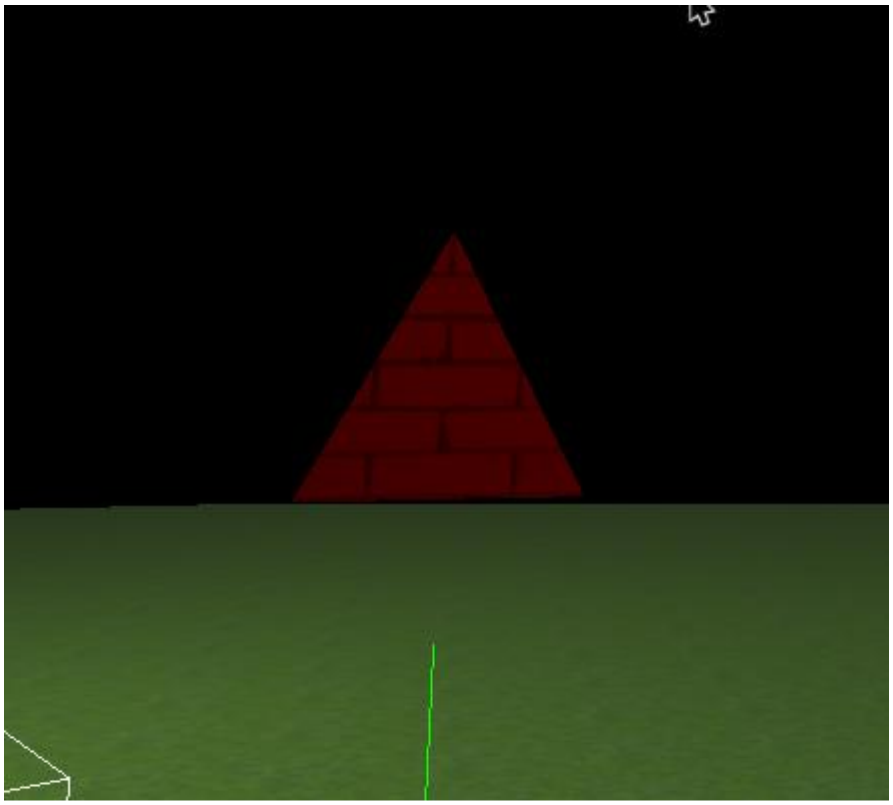
Next task in the lab was to give a behavior for the Pawn object. For this I selected RotateYBehavior. Following code snippet shows the pawn object addition and its behavior in TheGame's setup method.

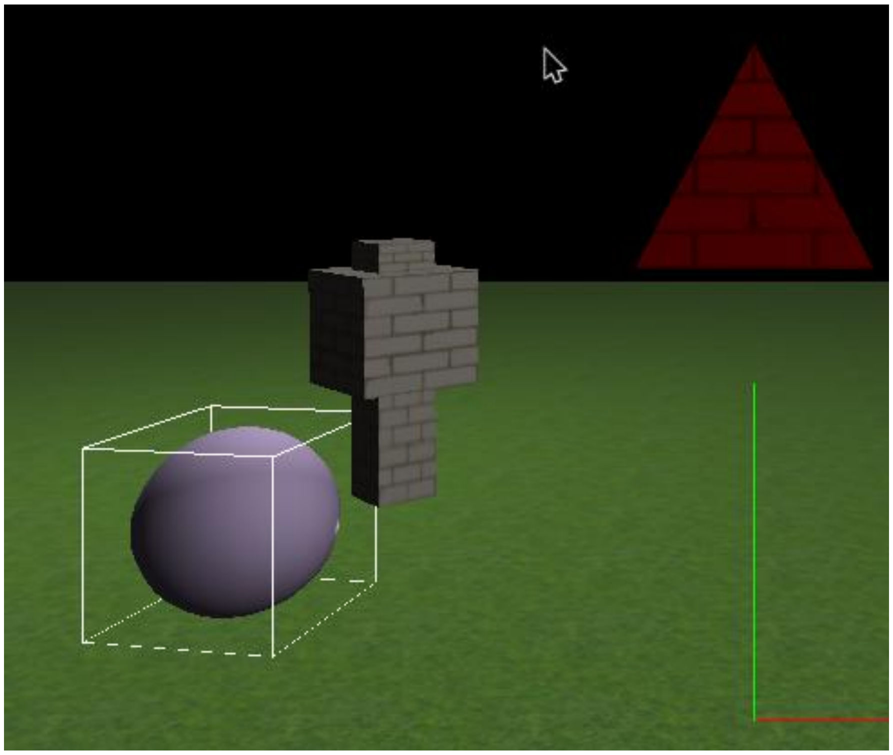
```
object = new Pawn();
object->setTexture(texture);
object->referenceFrame.setPosition(-3, 3, 0);
object->setBehavior(new RotateYBehavior(50.0f));
object->setShaderProgram(shader->getHandle());
objectMgr->addObject(object->getName(), object);
object->setVisibility(true);
```

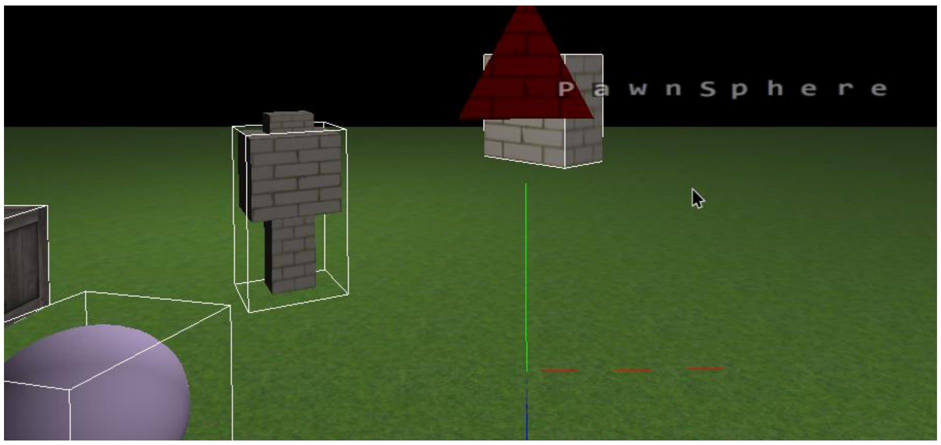
Next task was to add a bounding box to the Pawn. For this, I added a bounding box using the code for Crate's bounding box.

```
if (object) {
              TexturedCuboid* c = (TexturedCuboid*)object;
              object->boundingBox.set(1.1f, 1.1f, 2.1f);
              object->boundingBox.use = true;
              object->referenceFrame.setPosition(-3, 3, 0);
              object->referenceFrame.rotateWorldY(-45.0f);
              object->setBehavior(new BackForthBehavior(24));
              object->setSpeed(7);
              object->setVisibility(true);

              string name = c->getName() + "BB";
              object = new LineBox(
                     name, 1.1f + 0.01f, 1.1f + 0.01f, 2.1f + 0.01f);
              object->setShaderProgram(plainShader3d->getHandle());
              objectMgr->addObject(name, object);
              c->lineBox = (LineBox*)object;
              c->showBoundingBox = true;
       }
```

As seen from above code, I added the bounding box around the width, height and depth of the object, by adding a size 0.01 greater than the target object. Following screenshots show the triangle with texture, pawn object with RotateYBehavior and the Pawn with bounding box.






PawnSphere

Lab 12 (Collisions)

The objective of this lab was to learn to see collision in action. In addition to this lab also adds game assets externally.

I added Floor and Room assets using GameAssets.data file. Following code snippet shows the code to add Floor.

```
 <otfs>
  <name>
    Floor
  </name>
  <width>
    20.0f
  </width>
  <depth>
    20.0f
  </depth>
  <widthSegments>
    20
  </widthSegments>
  <depthSegments>
    20
  </depthSegments>
  <color>
    1.0f, 1.0f, 1.0f, 1.0f
  </color>
  <boundingbox>
    20.0f, 20.0f, 0.2f
  </boundingbox>
  <textureName>
    FloorTex
  </textureName>
  <shaderName>
    TextureShader
  </shaderName>
</otfs>
```

Then I added a textured cuboid in to the room again using the GameAssets.data file. This object is modified in the TheGame's setup method. As the final form I set the position of the crate (cuboid) object and set its visibility to true. As for the movement, I set the fixed in place variable to false. This causes the object to fall. The object falls from y position 5 to the ground.

```
       object = (OGL3DObject*)graphics->getGameObject("Crate");
       if (object) {
              object->referenceFrame.setPosition(0, 5, 0);
              object->setFixedInPlace(false);
              object->setVisibility(true);
       }

       object = (OGL3DObject*)graphics->getGameObject("Floor");
       if (object) {
              object->setGround(true);
```

```
	}
```

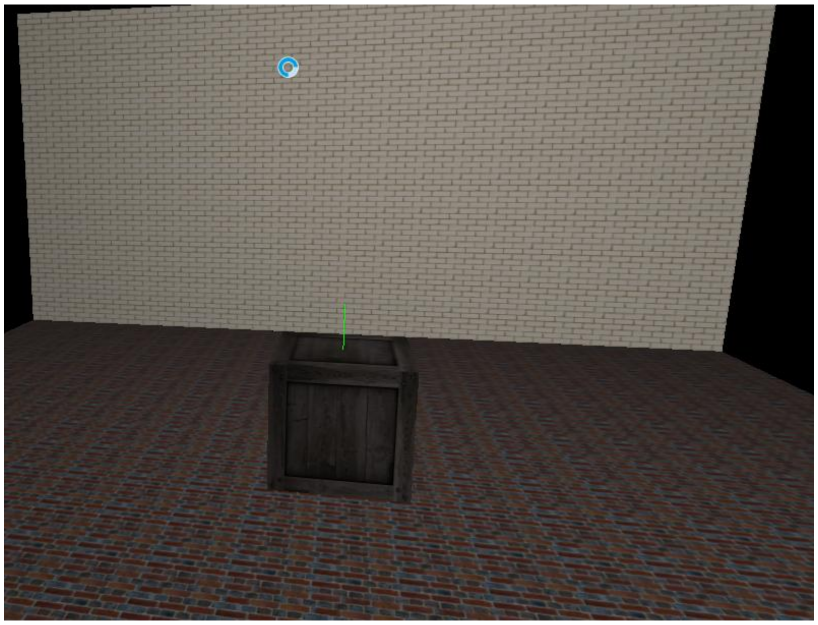
The velocity of the object is updated in GameObject's update method.

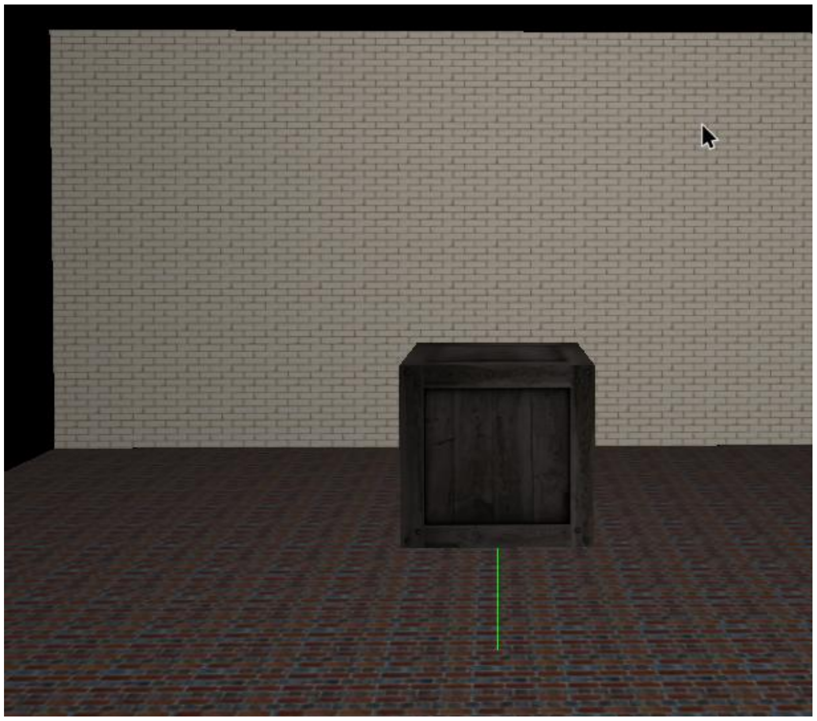
```
void GameObject::update(float elapsedSeconds)
{
	if (this->fixedInPlace)
		return;

	// Linear movement
	this->velocityDelta = this->velocity * elapsedSeconds;

	this->referenceFrame.move(this->velocityDelta);
	if (this->behavior) {
		this->behavior->update(elapsedSeconds);
	}
	this->velocity += this->acceleration * elapsedSeconds;
	this->velocity *= powf(this->damping, elapsedSeconds);
}
```

GameObject's resolveCollisions resolves the position of the abject after it hits the ground.

```
void GameObject::resolveCollisions()
{
	if (this->isColliding()) {
		this->referenceFrame.move(-this->velocityDelta);
		this->stopMoving();
	}
	else {
		if (!this->onGround) {
			this->setAcceleration(glm::vec3(0, -32.2f, 0));
		}
	}
}
```

The final output can be seen in following screenshot. The screenshots show textured room and floor. The crate (texture cuboid) is at the center which falls from a height. The second screenshot shows the falling crate. Once the create falls, it collides with the floor and the collision is resolved as discussed above.





Project 3

The objective of project 3 was to learn to create a 3D object with some animation.

Story

In project 3 I designed a truck that can be used to do surveillance or collect maps (futuristic version of Google's mapping car). The truck goes around a block of houses. Two cameras hover over the truck. Once the truck stops they go in opposite directions to collect data from the city. The truck again moves around the block. It then stops for the second time. This time, the cameras that finished collecting data come back and report the data to the device equipped inside the truck.

Construction

The truck is made up of two cubes joined together. The truck constructing method **generateBoxIndexedArrayTruck** uses 16 vertices to create the truck. This method is defined in ObjectGenerator class. Similarly methods to generate the cameras and the buildings are also defined in the ObjectGenerator class. A camera object is constructed using a pyramid over a cube. The pyramid uses 13 vertices. The buildings are created using cube creation method.

The room is constructed using flat surfaces. It is the combination of flat surfaces for Ground, Left Wall, Back Wall, Right Wall and Roof. In StockObjectLoader's loadObjects method these OGL3DObjects are added to the GameObjectManager object. As an example, following code snippet shows the creation of Ground flat surface.

```
object = new OGL3DObject("Ground");
	object->setIndexedArrayType();
	arr = ObjectGenerator::generateFlatSurface(10, 10, siVertexData[i].x,
siVertexData[i].z, { siVertexData[i].red, siVertexData[i].green, siVertexData[i].blue,
1.0f });
	object->setVertexData(arr.vertexData);
	object->setIndexData(arr.indexData);
	vboObject = OGLObject::createVBOObject(
		"triangles", arr.vertexData, arr.indexData, GL_TRIANGLES);
	object->addVBOObject(vboObject);
	gameObjectManager->addObject("Ground", object);
```

The camera objects are also created similar to how above objects are created. For creating the house different boxes are created one over another by setting proper position. A two story house is created using Lower half boxed array and upper half boxes array.

```
object = new OGL3DObject("House1_Lower");
	object->setIndexedArrayType();
	arr = ObjectGenerator::generateBoxIndexedArray(siVertexData[i].x,
siVertexData[i].y, siVertexData[i].z, { siVertexData[i].red, siVertexData[i].green,
siVertexData[i].blue, 1 });
	object->setVertexData(arr.vertexData);
	object->setIndexData(arr.indexData);
	vboObject = OGLObject::createVBOObject(
		"triangles", arr.vertexData, arr.indexData, GL_TRIANGLES);
	object->addVBOObject(vboObject);
	object->referenceFrame.setPosition(3.0f, 0.6f, -2.0f);
	gameObjectManager->addObject("House1_Lower", object);
```

The truck object is also created like other objects. However, as mentioned above it uses 16 vertices. The truck is added to the GameObjectManager object in StockObjectLoader class.

```
object = new OGL3DObject("Blue Truck");
	object->setIndexedArrayType();
	arr = ObjectGenerator::generateBoxIndexedArrayTruck(siVertexData[i].x,
siVertexData[i].y, siVertexData[i].z, { siVertexData[i].red, siVertexData[i].green,
siVertexData[i].blue, 1 });
	object->setVertexData(arr.vertexData);
	object->setIndexData(arr.indexData);
	vboObject = OGLObject::createVBOObject(
		"triangles", arr.vertexData, arr.indexData, GL_TRIANGLES);
	object->addVBOObject(vboObject);
	object->referenceFrame.setPosition(0, 0.1f, 0);
	gameObjectManager->addObject("Blue Truck", object);
```

This project externalizes various data. The window properties are stored in an external file called window.config. It contains basic window properties such as width, height and title. This file is read using the Configuration class. The attributes are set in WinMain function. Similarly the graphics data for each object is stored in a file called graphics.dat. It contains information such as width, depth, height and color values for each object created in the scene.

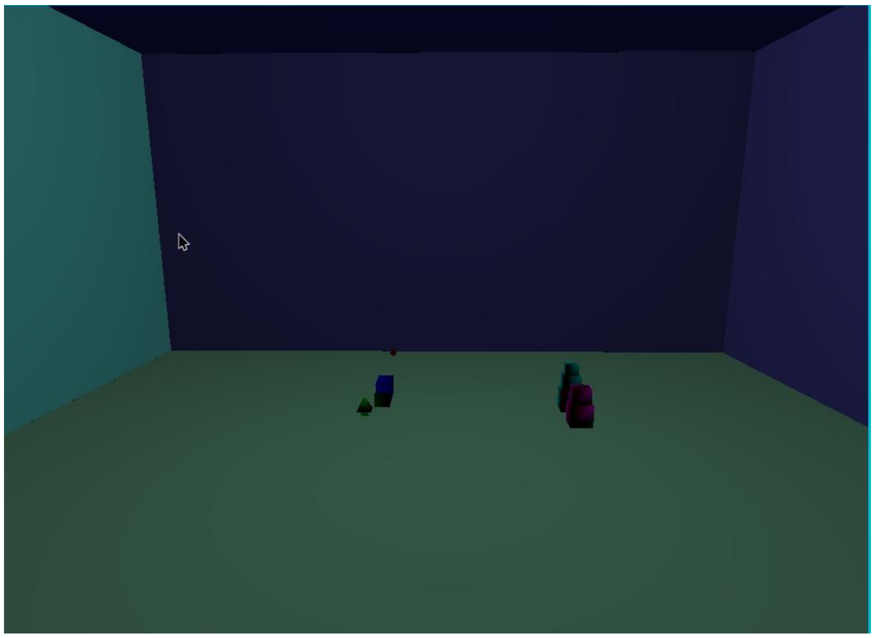
There are three classes that determine the animation for the graphics objects in the scene. The truck uses FourPointPatrolBehavior class for the block traversing animation. The cameras respectively use FourPointLoadUnloadBehaviorBackward and FourPointLoadUnloadBehaviorForward classes for their animation. These classes set up behavior in which the objects move forward following the object with FourPointPatrolBehavior, but when that object stops they move in different directions. These objects come to their original position, once the truck stops for the second time. Following code snippet shows the update method of FourPointLoadUnloadBehaviorForward class.

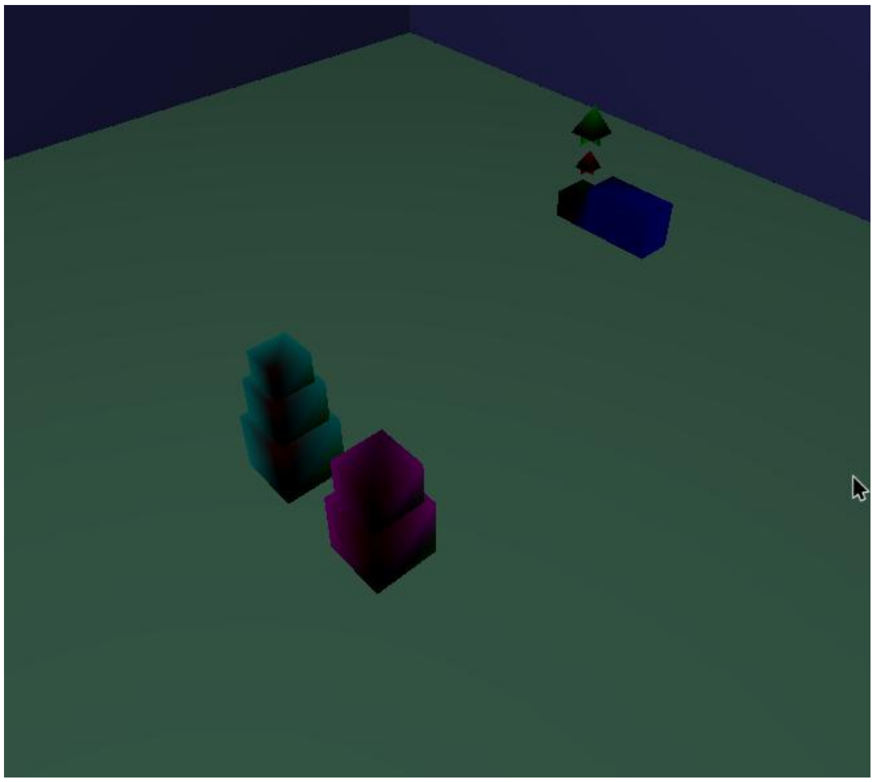
```
void FourPointLoadUnloadBehaviorForward::update(GameObject *object, float elapsedSeconds)
{
	OGLObject *obj = (OGLObject *)object;


	switch (this->state) {
	case MOVING_FORWARD:
	{
		delta = 5.0f * elapsedSeconds;
		this->distanceMoved += delta;
		this->totalDistanceMoved += delta;

		if (this->distanceMoved >= this->maxDistance) {
			this->state = TURN_AROUND;
			delta = this->distanceMoved - this->maxDistance;
			this->distanceMoved = 0;
		}
```

```
				obj->referenceFrame.moveForward(delta);

				// Check if the opposite block is reached
				if (totalDistanceMoved >= 2 * this->maxDistance) {
					this->state = STOP;
				}


				break;
			}

			case STOP: {
				float stopDelta = 5.0f * elapsedSeconds;
				this->distanceStopped += stopDelta;

				if (this->distanceStopped >= this->distanceStop) {
					this->state = MOVING_FORWARD;
					this->totalDistanceMoved = 0.0f;
					stopDelta = this->distanceStopped - this->distanceStop;
					this->distanceStopped = 0;
				}

				obj->referenceFrame.moveForward(stopDelta);

				break;
			}



			case TURN_AROUND: {
				float rotateDelta = 90.0f * elapsedSeconds;
				this->degreesRotated += rotateDelta;

				if (this->degreesRotated >= this->maxRotation) {
					this->state = MOVING_FORWARD;
					rotateDelta -= this->degreesRotated - this->maxRotation;
					this->degreesRotated = 0;
				}

				obj->referenceFrame.roateLeftWithForwardMovement(rotateDelta, delta * 
0.5f);
				break;
			}



			}

	}
```

As can be seen from above code the code uses three states (MOVING_FORWARD, STOP and TURN_AROUND). The behaviors are done just like done for previous animation labs. The truck moves forward at it enters the MOVING_FORWARD state. Once the object reaches the maximum length of one side, it turns around and enters into TURN_AROUND state. It turns around by both moving forward and rotating along the Y-axis. These two movements give it a smooth transition. After it turns around it again transitions to MOVING_FORWARD state. In the MOVING_FORWARD state another check is made to see whether the truck has moved twice the maximum distance (i.e.; whether it has moved to opposite sides of its path). If it has moved twice the maximum distance the truck stops. Thus it transitions to the STOP state. During stop state it basically slows down the movement to a very small distance over the course of given time. Once that distance has been passed it again moves forward.

The output can be seen in following screenshots. The screenshots show the vehicle and the cameras. In the second screenshot the cameras hover over the truck and in the first screenshot the cameras are travelling towards the truck as it stops at the corner.





Project 4

The objective of project 4 was to learn to create objects using hierarchical model using stack. This project also focuses on lighting.

Story

In project 4, I designed two rooms. These rooms have two light sources at the ceiling (each room has its own light source). The room consists of a ceiling, a floor and four walls. One of the walls has door. Each of the door leads to another room. Each of the rooms is furnished with a table and few chairs. There is a player (Tank/Turret) in the room that keeps on patrolling between the rooms. When the player (now camera) enters a room the light on the previous room turns off and the light on the entering room turns on.

Construction

The room is constructed using Walls. Each wall is constructed using a cuboid object, which has a very small height. The floor of the room is considered as the reference of the room and walls and ceiling are constructed using the stack object. The wall with door is a special type of wall that is constructed using three cuboids. The cuboids are stacked one over another, leaving the space for the middle of the room, which consists of the door. The Room class's render method uses reference frames to construct the room and add furniture.

```
void Room::render()
{
	this->floor->referenceFrame = this->referenceFrame;
	this->floor->referenceFrame.translate(0, -2 * this->thickness, 0);
	this->frameStack.setBaseFrame(this->floor->referenceFrame);
	this->floor->render();


	this->frameStack.push(); // Floor
	{
		// North Wall
		this->frameStack.rotateX(90.0f);
		this->frameStack.translate(0, -this->width/2, -this->height/2);
		this->northWall->render(this->frameStack.top());

		// South Wall
		this->frameStack.rotateZ(180.0f);
		this->frameStack.translate(0, -this->width, 0);
		this->southWall->render(this->frameStack.top());

		// West Wall
		this->frameStack.rotateZ(90.0f);
		this->frameStack.translate(this->width / 2, -this->height / 2, 0);
		this->westWall->render(this->frameStack.top());

		// Ceiling
		this->frameStack.rotateX(90.0f);
		this->frameStack.translate(0, -this->width / 2, -this->height/2);
		this->ceiling->render(this->frameStack.top());

		this->frameStack.push(); // Ceiling
		{
			this->frameStack.translate(0, 0, 0);
			this->lightSourceObject->render(this->frameStack.top());

			this->frameStack.translate(0, 0.375, 0);
			this->lightSourceSecondObject->render(this->frameStack.top());
		}
		this->frameStack.pop();

		// East Wall
		this->frameStack.rotateX(90.0f);
		this->frameStack.translate(0, -this->depth / 2, -0.25*this->height);
		this->frameStack.rotateX(180.0f);
		this->eastWall->render(this->frameStack.top());

		// Table
		this->frameStack.translate(0, 0, 0);
		this->frameStack.rotateX(90.0f);
		this->frameStack.translate(0, this->height - 3.0f, 0);
		this->frameStack.rotateY(90.0f);
		this->frameStack.translate(this->width / 2, 0, 0);  // At this point the 
reference is pointing at the center of the room

		this->frameStack.translate(0, 0, -this->depth / 2 + 2.0f);
		this->table->render(this->frameStack.top());

		// Chair
		this->frameStack.translate(2.0f, 0, 0);
		this->frameStack.rotateY(180);
		this->chair->render(this->frameStack.top());

		// Chair 2
		this->frameStack.translate(4.0f, 0, 0);
		this->frameStack.rotateY(180);
		this->chair->render(this->frameStack.top());
	}
	this->frameStack.pop();
}
```

As seen from the code above, the base frame is set to the floor. From there various walls are added and finally the furniture (one table and two chairs) is added to the room.

To create one of those walls a Wall class is used. To keep track of the wall's orientation, I drew axes. These axes are drawn only when DRAW_AXES constant is set to true.

```
void Wall::render()
{
	this->surface->referenceFrame = this->referenceFrame;
	this->frameStack.setBaseFrame(this->surface->referenceFrame);
	this->surface->render();

	if (DRAW_AXES) {
		this->frameStack.push();
		{
			// Axes
```

```
				this->frameStack.translate(0, this->thickness, 0);
				this->axes->render(this->frameStack.top());
			}
			this->frameStack.pop();
		}
}
```

A turret object moves between two rooms. The turret object is created using previous lab code. The room has light sources at the ceiling, which are created by stacking two cuboids one over another to give a feeling of a dome light. The light object is attached at the position where these cubes exist.

The furniture (chairs and table) are created using several cuboids. Each furniture object has its own class that maintains its own stack frame. These objects are then added to the room at the floor.

This project externalizes various data. The window properties are stored in an external file called window.config. It contains basic window properties such as width, height and title. This file is read using the Configuration class. The attributes are set in WinMain function. Similarly the graphics data for each object is stored in a file called graphics.dat. It contains information such as width, depth, height and color values for each object created in the room.

BackForthWithLight class gives the turret object it patrolling.

```
void BackForthWithLight::update(GameObject *object, float elapsedSeconds)
{
	if (this->leftLightSource != NULL && this->rightLightSource != NULL && this-
>camera != NULL) {
		OGLObject* obj = (OGLObject*)object;
		float delta = 5.0f * elapsedSeconds;
		this->distanceMoved += delta;
		glm::vec3 cameraPosition = this->camera->getPosition();


		switch (this->state) {
		case MOVING_BACKWARD:
			if (this->distanceMoved >= this->maxDistance) {
				this->state = MOVING_FORWARD;
				delta = this->distanceMoved - this->maxDistance;
				this->distanceMoved = 0;
			}

			obj->referenceFrame.moveBackward(delta);
			break;
		case MOVING_FORWARD:
			if (this->distanceMoved >= this->maxDistance) {
				this->state = MOVING_BACKWARD;
				delta = this->distanceMoved - this->maxDistance;
				this->distanceMoved = 0;
			}

			obj->referenceFrame.moveForward(delta);
			break;
			}
		}
	}
```

The LightBehavior class’s update method defines the light source on and off behavior based on the camera position. For this to work, the left and right light source objects are passed to the LightBehavior class along with the OGLFirstPersonCamera object. LightBehavior’s update method gets reference to the room. Based on the camera position and the room position it switches between the lights by setting their intensity to 0.5 or 0.

```
void LightBehavior::update(GameObject *object, float elapsedSeconds)
{
	if (this->leftLightSource != NULL && this->rightLightSource != NULL && this-
>camera != NULL) {
		OGLObject* obj = (OGLObject*)object;

		glm::vec3 cameraPosition = this->camera->getPosition();
		glm::vec3 roomPosition = obj->referenceFrame.getPosition();

		if (abs(cameraPosition.x - roomPosition.x) < 5) {
			this->leftLightSource->setIntensity(0.5f);
			this->rightLightSource->setIntensity(0.0f);
		} else if (abs(cameraPosition.x - roomPosition.x) > 5 &&
abs(cameraPosition.x - roomPosition.x) < 10){
			this->leftLightSource->setIntensity(0.0f);
			this->rightLightSource->setIntensity(0.5f);
		}


	}
}
```

Output Screenshots