Final Project Report

A Report Presented to
The Department of Computer Science & Software Engineering
Concordia University

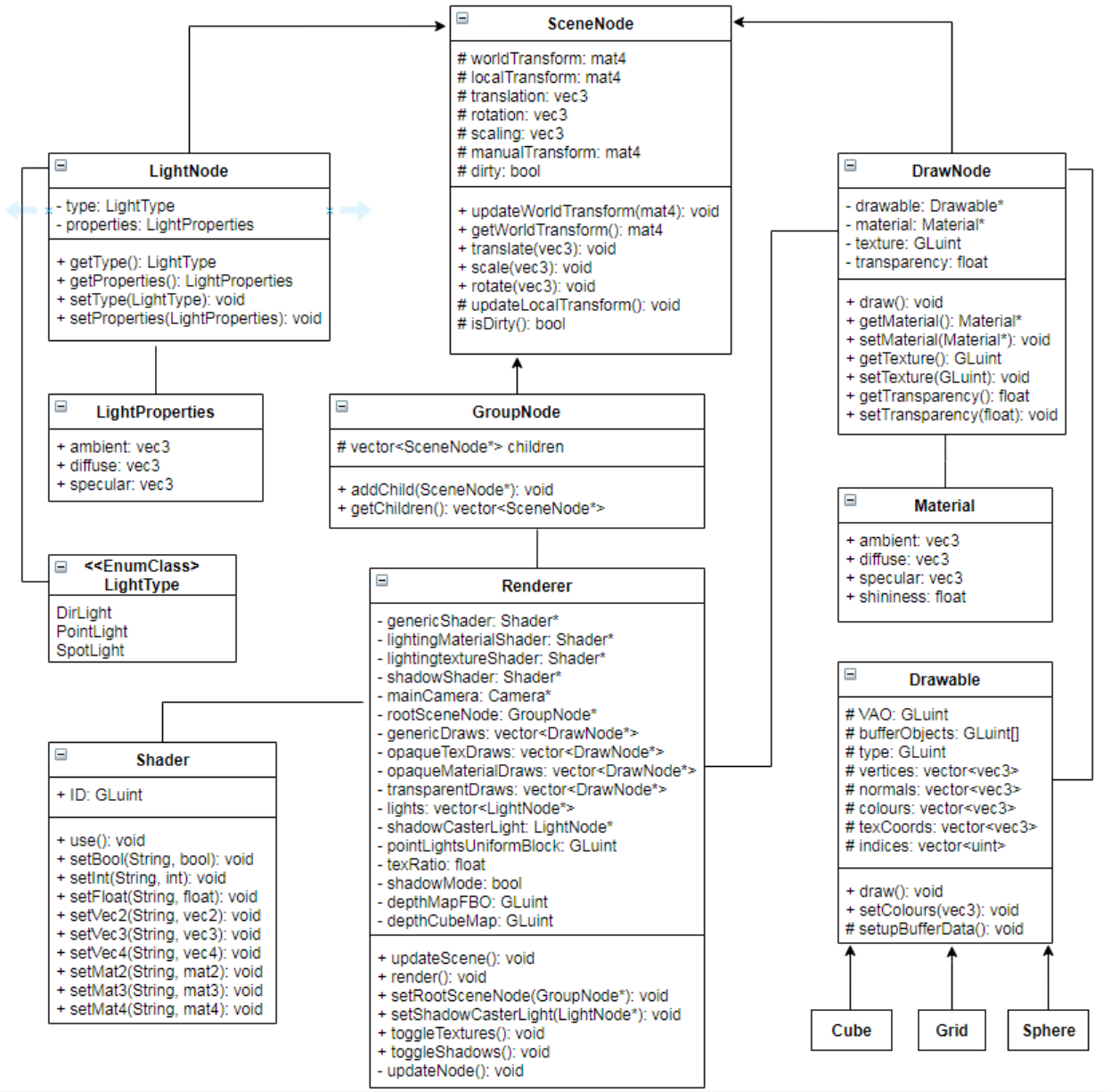
In Fulfilment
of the Requirement
of COMP 371

by
Dan Raileanu (ID: 40019882)
Mohamed-Yasser Houssein (ID: 27650922)
Mohd Tanvir (ID: 40014010)
Muherthan Thalayasingam (ID: 27223064)
Radhep Sabapathipillai (ID: 40033092)

Concordia University
August 2020

Starting Point

After Assignment 2, we had the following program structure to work with.
Note that some non-important methods are not shown for simplicity. Also the Model class used to create the Student Models in assignments is not included since not needed for project.



We have a good Hierachical Model in the form of SceneNode and classes that inherit from it.
-SceneNode has the default implementaiton for spacial transformations that all nodes inherit.
-GroupNode is only class allowed to have children, thus any complex Model should inherit it.
-DrawNode is only class allowed to have a Drawable geometry and has a draw().
-LightNode holds the required parameters for lights. Currently only supports PointLights.
-Drawable has the Buffer Objects to contain all the geometry and a draw() that has OpenGL draw it. Everything that needs to be drawn has to inherit form it.

The Renderer class has a low cohesion, as it does too much. It currently handles traversing the tree from the rootSceneNode, updates nodes' spacial transforms and sorts all nodes into appropriate vectors. It sorts DrawNodes in different categories and each requires a different shader to be drawn. Note that only transparentDraws absolutely require to be sorted appart, as they have to be sorted back to front and drawn at the very end, but we decided to sort all of them to reduce number of shader swaps. It also allows us the option to sort the DrawNodes dependig on texture/material in order to reduce setting uniforms between draw calls.

Planning

We decided to go with the Rubik Cube option. The main reason is that we have a good Hierarchical Modeling and are most familiar on how to manipulate objects in space, which seems to be the main objective in designing a rubik cube.

What we plan on implementing:

- Basic rubik cube that simply has different colours on each face (counts as 1 puzzle).
- Inherit from base RubikCube and add texture implementations. One with where each mini-cube of a face has a texture of some geometric shape. The second as a jigsaw puzzle where a texture is split into 9 parts for each mini-cube of the face.
- Inherit from base RubikCube and add particles implementation. Particle colors will represent the different colors of a rubik cube.
- Display timer. Have a 3D Model made of digits in TimeX format that change in such a way to increment timer every second. Also have to be able to reset and pause it.
- Create SkyBox
- Create an arcade ambiance by having some flashy animations, like lines going around or some shapes doing something interesting. Even better if procedural.
- Create some animation around the timer to give arcade feeling
- Audio

Implementation

Dan Raileanu

➔ **Basic Rubik Cube implementation:**

First I decided what functionality the rubic cube should have:
-Select a face, so that we only have to specify if rotation is to be clockwise or counter-clockwise.
-Have something to indicate clearly which face is selected.
-Be able to rotate only the part that is selected. It should also rotate over several frames and not simply change in 1 frame.
-Be able to rotate the whole rubic cube by 90 degrees CW/CCW on any of the 3 axes.

Problem: The rubic cube will rotate by 90 degrees on all 3 axes and the current implementation for rotations uses Euler angles, which will easily result in Gimbal Lock.
Solution: Updated SceneNode class to use Quaternions instead of Euler angles.

Selecting Face

First I created the rubic cube from 26 mini cubes (didn't incluse the one at the center as it's never seen displayed). The rubic cube inherits from GroupNode and has all the 26 cubes as children, positioned appropriately around the center. Selecting a face should have a mechanism to get a handle on all the cubes that will need to be rotated, so I used std::map<std::tuple<int,int,int>, DrawNode*> to have a tracking of all the mini-cubes in the rubic cube. The 3 ints corespond to the coordinates of a cube (DrawNode) going from (1,1,1) to (3,3,3). Selecting a face then is then simply going on a double for-loop and selecting all the appropriate DrawNodes and placing them inside an std::vector<DrawNode*>, which will be used later to perform the rotation. In order to clearly indicate which face is selected, I created a Pyramid drawable and have a DrawNode member that will have its position/rotation be updated whenever we select a face.

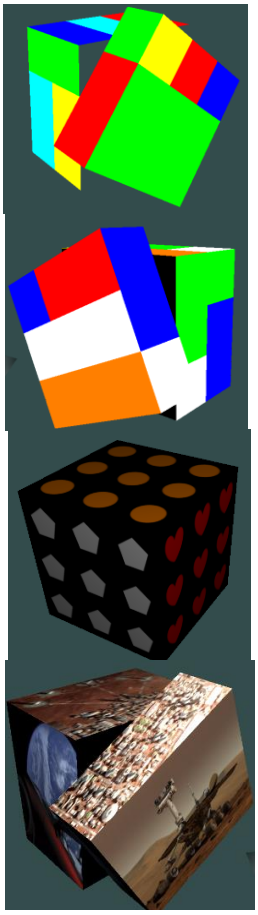
Animation

The animation of a rotation is implemented by having rotateCW/CCW set a boolean to know that we are in an animation. This way, further calls to the function won't do anything until rotation is completed. These functions then use a switch case and according to which face is selected: set a glm::vec3 to have the rotation angles that the selected nodes from earlier are to be rotated. Since the rotating cubes will change coordinates, they also update the std::map accordingly to have the updated coordinates at the end of the animation.

I modified SceneNode to have a virtual Update(float dt) function that by default simply calls updateWorldTransform(), but could be overriden to do more. The rubic cube will check if an animation is to be performed and if so call animationUpdate(). AnimationUpdate() uses dt and the rotation vector set earlier to perform the rotation on the selected cubes. When the animation is complete, it sets the animation boolean to false to allow a new rotation to be performed by the rotateCW/CCW functions.

For the rotation of the whole cube along an axis, I didn't want the selected face to rotate as well. It's more intuitive and easier for a user to select a face according to global axes than to local ones, as it's not easy to track which one is which as it's being rotated around. I added 6 functions called rotateX/Y/Z(CW/CCW) which are similar to the previous rotateCW/CCW, then a similar update function for the animation of the whole cube. The only difference is that we also update the selected cubes according to which face was selected before the rotation.

➔ **Rubic Cube with Textures**



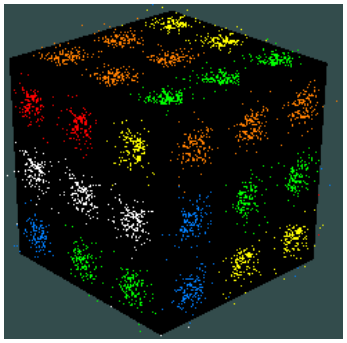
Problem: Most cubes will need 2 or 3 textures, and because they are rendered in a single draw call, we cannot use different textures for each side. Shaders can take many textures, but they there is no mechanism to select a specific texture for a specific side of a cube.
Solution: A texture atlas could have been a solution, but I decided against it. Instead I modified the RubicCube class to have GroupNode instead of DrawNodes, where each group contains several Quads arranged such that it looks like a cube.

I only created the Quads visible from the exterior, but it caused a problem where you could see the background when it would rotate. To fix it, I added an additional black Quad behind every exterior quad, thus giving the appearance that the cube is black from the inside.
RubicCubeTextures loads textures and sets appropriate ones for each Quad. RubicCubeJigsaw imeplements textures, but such that 1 texture is shared between different Quads. This is done by manually setting the texture coordinates of Quads such that they only display a portion of the whole texture.

➔ **Rubic Cube with Particles**

For the last kind of rubic cube, I wanted each Quad color to be represented by particles of the given color. Every drawable geometry we have thus far uses it's own VAO and requires 1 draw call. The grid is the largest drawable with 20 000 vertices, but it was easy to set the positions and they are never modified. I needed a ParticleEffect class to inherit from Drawable in order to attach it to a DrawNode, which would be attached to the rubic cube. However, the class should implement all the required behaviour to update the particles at every frame. I used this article as reference on how to implement a

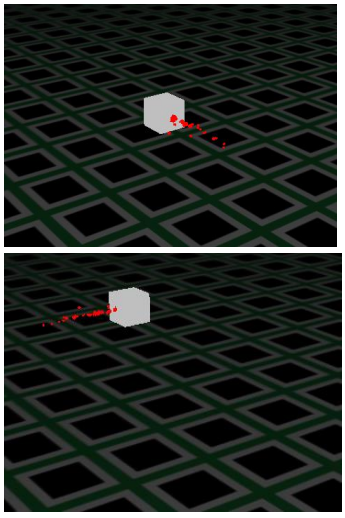
particle system: https://www.3dgep.com/simulating-particle-effects-using-opengl/#Taking_full_advantage_of_the_power_of_the_GPU
While some code was copied, I had to modify it a bit as they used depreciated opengl and it had to fit with the current implementation of renderer. Even though we normally implement particles as quads, I had to imeplement it as cubes.The main reason was that these particles would be attached to a DrawNode, which would be rotated around and not allowing me to keep the quad facing the camera at all times. I read on the concept of Billboards to fix this issue, but the current imeplementation of Renderer would have to be heavily modified for it to work and I didn't have the time. The difference in performance wasn't too bad, allowing me to render thousands of cubes with only about 20 fps drop on a weak 2GB VRAM PC.



The basic idea behind ParticleEffect class is that it allocates a very large vertex buffer capable of holding all vertex/color/index data for MAX_PARTICLES particles. Particles have position, velocity, color, rotation angle, age and lifetime. ParticleEffect has an update function that is called on every frame, which goes through all the particles and if age>lifetime, it calls ParticleEmitter to generate a new one. If age<lifetime, it updates the position according to velocity vector. Then it calls glBufferSubData to update the data on the GPU. Since we are updating this data very often, we set the GL_DYNAMIC_DRAW hint when first called glBufferData.

With a working particle system, we simply attach it to the Rubic Cube as a Drawable for a DrawNode.

➔ **Decorative Cubes**

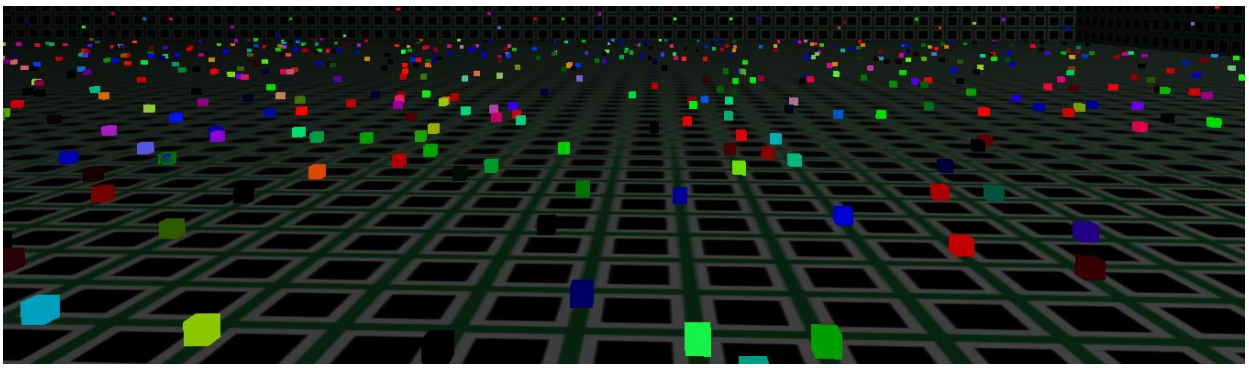


We had to keep the grid as we needed some sort of surface to show that shadow lighting works. However, a simple grid was boring, so I wanted to have some procedurally animation to give a more arcade feeling. My first idea was to have cubes travel randomly along the lines of the grid while shooting particles from the back. However, once they would turn 90 degrees, the particles would also immiediately rotate, which wasn't too nice. I could have an animation where they slowly rotate before moving again, but I realized that I would need hundrends of these cubes, each with hundreds of particles attached in order to have anything esthetic. Inspired from the ParticleEffect implementation, I decided to create another Drawable capable of holding a large ammount of cubes and update them in such a way that they travel along the grid lines and randomly turn sometimes, while also changing colors. At first, I implemented this as Quads, as I didn't have the issue where they rotate, but it wouldn't look too good when looking at them from a low angle, thus I made them

as cubes instead. The cubes are very small and I added a function to dynamically add/remove cubes at runtime.

➔ **Audio**

Finally, some sound was needed to provide an arcade ambiance. I used the sound engine



provided by IrrKlang library from https://www.ambiera.com/irrklang
I also used an open source sound track from https://www.bensound.com and https://freesound.org
The full link is in the main file. Addded a song to play on repeat and a sound effect to play when rotating the cube.

Mohamed-Yasser Houssein

➔ **Timer class**

The timer class contains 3 main user accessible functions:

void start():

The function first records the processor time by a call to *clock()* which is a function available to us by C++. Then we assign that time to an unsigned long variable called *begTime.*

Void timeUpdate(unsigned long timeElapsed,bool timeStarted):

This function takes in as arguments the timer's elapsed time since last frame render and the boolean value that indicates whether the timer is paused/not started yet. If the time has not started yet or that it is on pause, then *timeStarted* will be false and nothing happens. If the boolean is true, then we compare the elapsed time in seconds with the previous second passed. If we have gained a second, then the function calls of a UI update of the timer.

void time elapsedTime():

the elapsed time function returns the time elapsed in seconds by dividing the processor time by the global CLOCK_PER_SEC which is also made available to us by C++.

void pause():

This function sets the *timeStarted* variable to false. The key is to also reset the current_seconds variable to zero as well otherwise when the timer is restarted, the time will not update until the elapsed time catches up to the previous saved second.

void reset():

This function resets the UI to all zeros and resets their corresponding int values to zeros as well.

Helper functions

Other functions, although they are public, are simply methods to be used privately.

To sum up the story behind the private functions, When a second has elapsed, the *timerUpdate* function is only concerned in updating the right second. It does this by making a call to *updateRightSecond()*. Once this function gets called, a ripple effect is created where every digit updates the digit to its left once it reaches a certain value(0 or 6).

Challenges

My main challenges where drawing the timex models and also deconstructing the models and constructing them again when there is a digit update. With the help of my teammates, I was able to complete those challenges. For the models, my teammate has proposed the idea of drawing the digit 8 and then utilize this digit to construct all other models. Moreover, when it came to deconstruct the models, my teammate pointing out that there was a memory leak from the way the models are being deconstructed and proposed to use other available methods such as *removeChild* followed by *delete.*

Mohd Tanvir

I had initially started working to create the skybox for the project (Assignment 3). Since I had initially created some form of skybox for the quiz 2, I tried to replicate the same form. But what I had come to realize is that the way I had previously created the skybox had some flaws. One of my teammates had then suggested using the form that is shown on the site learnopengl.com. We then proceeded on following those steps where we created a cubemap and then later created the skybox based on the cubemap. Near the end we realized that there was an issue with the texture of the skybox and the shadow/light, so we had to pivot a bit to fix everything.

Another part that I was working on in the project with another teammate was creating dynamic border lines around the timer. The initial plan was to have border lines surround the timer and have it rotated in a clockwise manner. The issue we faced was having the lines curve in a

smooth transition near the edges. We then later changed the transition and had a different set lines with the timer being the parent node that surrounded the timer.

Radhep Sabapathipillai

➔ **Skybox Attempt**

Initially we wanted to create a skybox that had lights all around it. I and a team-mate followed a tutorial that was found on LearnOpenGL and we were able to create it. However, we noticed that there was an issue that created large black shadows on the surface of the plane. We tried many solutions but were unable to resolve it. We later conceded that due to the time left for the project, we would not be able to implement the skybox and therefore we moved to our next task.

➔ **Timer Lines:**

I and a team-mate wanted to create lines that would go around our timer. The lines were meant to go around in a snake formation where they would go straight and then curve at the edges if needed. This was initially planned using multiple cubes to make up a single line and then rotating one cube at a time around the edges. However, we found that this would cause severe framerate issues as we would be rendering many cubes. An idea that we had to resolve this was rather to use two lines for each single line. The line would have a head and a tail. When it came close to an edge, we would scale down the head line and rotate it while simultaneously scaling up the tail line to make it look as if it were a smooth transition. This came with many complications as we were not able to seamlessly transition these lines around the edge. It was abrupt and we could not figure out a way to resolve this. Due to these unforeseen issues, we decided to do an alternative line approach which was to criss-cross two lines that were below and above the timer. We made sure that these lines were synced but started at the opposite sides. This gives a nice effect and we believe it is a good alternative to what we wanted to implement initially.

Muherthan Thalayasingam

➔ **Timer Attempt:**

We needed to figure out a way to have a timer model that would start a timer when we begin our Rubik's cube puzzle and that would end as soon as we solve it. Of course, this meant figuring out the logic for the trigger and end events and to visualize that in a way that would make sense. The challenge here was not the logic as that was quite simple to write in C++, but rather it was to figure out how the timer would be shown in 3D. In other words, we want to show the timer update in real time with the numbers changing in real-time using our 3D models.

Approach Considered:

The starting approach was to take was to first write out the logic that can start a timer. And then, the goal was to use the model class we have used in Assignment 1 and 2 to make a model for digits 0 through 9. With every second that passes by, I was going to change the model in real-time by changing the displaying model with the next and so on. So as the timer goes for 00:00 to 00:01, I would change the last model from the model for digit 0 to model for digit 1, and so on. However, I realized that this may be a slow process as I am constantly having to remove one model and adding another. My teammates have figured out a better solution that provides a more efficient way to display the 3D timer – this was discussed further in their implementation.