[CG] Project Report (Phase 3)

Guilherme Sampaio A96766 · Luís Pereira A96681 · Rui Oliveira A95254 · Tiago Pereira A95104

May 2023

1 Introduction

The following report refers to the group project of the Computer Graphics class of 2023, authored by Guilherme Sampaio, Luís Pereira, Rui Oliveira and Tiago Pereira. These elements make up group 60.

The goal of the project is to create a simple graphics engine using OpenGL and GLUT in C++. It is divided in four checkpoints: this report refers to the third one only. In this checkpoint, vertex buffers objects were to added to the project, as well as animations using Catmull–Rom curves and Bezier patches. To demonstrate this new functionality, the model of the solar system was updated to have animations, as well as a teapot generated from a patch serving as a commet.

As a result, aside from the usual section on how to run the project, this report will be divided into three main sections: one about the implementation of the vertex buffer obejcts (VBOs), another on the implementation of animations, and the other about the making of the Bezier patches.

2 Running the project

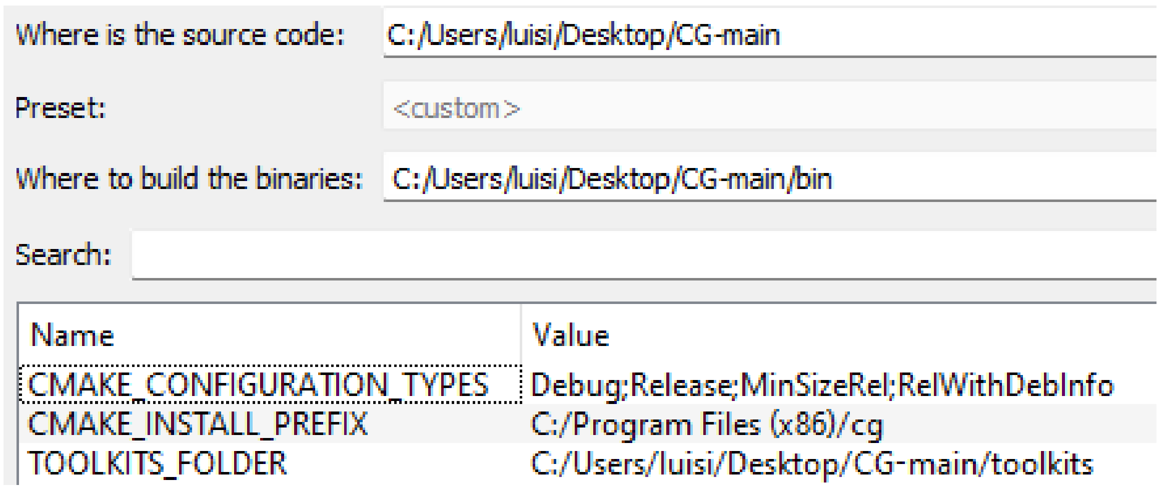
2.1 UNIX

Nothing has changed from the previous phase. Compile the project by running make and run it as the statement proposes.

2.2 Windows

Nothing has changed from the previous phase. To run the project in Windows, a CMakeLists.txt file was developed. All toolkits needed are included under toolkits/. The configuration is shown in Figure 1. In essence, the "*Where is source code*" should be the root of the project, *Where to build the binaries* the bin/ folder, and TOOLKITS_FOLDER the toolkits/ directory.

After configuring and generating, the Visual Studio project should be good to go. After building the solution, the project can be run in similar fashion to the UNIX version. The executables are placed under bin/Debug. The engine is called cg.exe and the generator generator.exe. All arguments are the same as their UNIX counterparts.



Where is the source code:	C:/Users/luisi/Desktop/CG-main
Preset:	<custom>
Where to build the binaries:	C:/Users/luisi/Desktop/CG-main/bin
Search:	

Name	Value
CMAKE_CONFIGURATION_TYPES	Debug;Release;MinSizeRel;RelWithDebInfo
CMAKE_INSTALL_PREFIX	C:/Program Files (x86)/cg
TOOLKITS_FOLDER	C:/Users/luisi/Desktop/CG-main/toolkits

Figure 1: CMake Configuration

3 Vertex Buffer Objects

In this fase the group was to implement VBOs into the project. The goal of this optimization is to minimize time spent transmiting data between CPU and GPU by storing some data directly in the GPU. In order to achieve this, the GLUT extension GLEW, which provides useful functionality for interacting with the GPU.

In order to adapt the project, minimal structural changes were needed. A new Shape::initialize() method was created to copy the shape data to a VBO and the Shape::draw() method was changed accordingly. The group also thought it would be interesting to create a Shape::cache to avoid parsing the same files multiple times, and found this to be the right time to implement it.

Currently, the project doesn't use indexed VBOs, though the group has plans to chage that by the next (and last) phase of the project.

4 Animations with Catmull-Rom curves

In order to support animations, the draw methods of the existing class Rotation and the new class CatmullRom are now time dependant, and both classes receive the period of the animation in their constructor. Although feasible, the group didn't implement any non-periodic animations, and therefore left classes Translation and Scale unchanged.

In order to calculate the position in the Catmull-Rom curve, first the segment of the curve is determined. Then, the four control points of the segment are multiplied with the Catmull-Rom matrix and the time row vector to obtain the position vector. The velocity vector is calculated in the same way by simply using the derivative of the time row vector instead.

The value of the position is then used to perform a translation. If the xml element defines `align="True"`, the velocity is then used to rotate accordingly. In this case, the instance of CatmullRom stores the up vector between translations to allow the alignment to be calculated (it starts as 0, 1, 0).

5 Bezier Patches

Generating 3D models based on provided Bezier patches has two main stages. Firstly, the contents of the file containing the patches is read and parsed. Secondly, the patches are converted into triangles. The first stage is rather trivial and uninteresting. The second stage is much more envolved and complex, so that is what we will be focusing on.

For each patch, we will compute its points using the formula [1]:

$$p(u,v) = \begin{pmatrix} u^3 & u^2 & u & 1 \end{pmatrix} M \begin{pmatrix} p_{00} & p_{01} & p_{02} & p_{03} \\ p_{10} & p_{11} & p_{12} & p_{13} \\ p_{20} & p_{21} & p_{22} & p_{23} \\ p_{30} & p_{31} & p_{32} & p_{33} \end{pmatrix} M^T \begin{pmatrix} v^3 \\ v^2 \\ v \\ 1 \end{pmatrix} \tag{1}$$

where p_{ij} are the given control points of the patch, $u, v \in [0,1]$ and

$$M = M^T = \begin{pmatrix} -1 & 3 & -3 & 1 \\ 3 & -6 & 3 & 0 \\ -3 & 3 & 0 & 0 \\ 1 & 0 & 0 & 0 \end{pmatrix}$$

From 1 a couple of observations arise:

1. As $u, v \in [0,1]$ and we need to divide each patch in a given level of tesselation (let's call it D), we will need to compute $p(u,v)$ for all

$$u, v \in \{0, \frac{1}{D-1}, \frac{2}{D-1}, ..., \frac{D-2}{D-1}, 1\} = \{\frac{n}{D-1} | n \in \{0, 1, ..D-1\}\} \tag{2}$$

2. The elements of the matrices are of different types: Most of the matrices are real, with the exception of the one containing the control points of the patch, whose elements are points in 3 dimensional space.

To resolve the problem that arises from the previous observation, we need to define multiplication between a 3D point and a real number, alongide addition between two 3D points. The natural definitions for these are

$$\forall x \in \mathbb{R}, y = \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} \in \mathbb{R}^3, x \cdot y = y \cdot x = \begin{pmatrix} x \cdot y_1 \\ x \cdot y_2 \\ x \cdot y_3 \end{pmatrix} \tag{3}$$

$$\forall x = \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix}, y = \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix} \in \mathbb{R}^3, x + y = y + x = \begin{pmatrix} x_1 + y_1 \\ x_2 + y_2 \\ x_3 + y_3 \end{pmatrix} \tag{4}$$

Implementing this operations in C++ is as simple as declaring and implementing:

```
Point operator *(float x, Point p);
Point operator *(Point p, float x);
Point operator +(Point p1, Point p2);
```

Now we can multiply these matrices. However, the difference in types mentioned earlier means it would be needed to either implement a function for each combination of types, or a generic matrix multiplication function would need to be created. The latter option was picked, as it was the most flexible, in spite of the additional complexity.

The generic function designed for this is void matrixProd(int n, int m, int l, T∗ a, S∗ b, U ∗c) declared in utils .hpp. This function works if there is addition for elements of C, and multiplication between elements of type T and S. The only nuance when implementing this function was defining the default value for type U. Normally, i.e., in $\mathbb{R}$, that value is zero. In $\mathbb{R}^3$, that value would be $(0, 0, 0)$. To generalize, elements of the output matrix are initialized with `U{}`, which works as expected [2].

Finally, with the points computed, the problem is reduced to grouping these to form triangles. We chose to group them in squares and then generate the triangles using the previously implemented generateSquare function. For each patch, a point will form a square with the points immediately next to it and below it, like shown in Figure 2. If we index the points based on the u and v values used to generate them, point p_{ij} would be grouped with $p_{i+1,j}$, $p_{i+1,j+1}$ and $p_{i,j+1}$ (if such points exist). The points are passed to the generateSquare function in clockwise order.

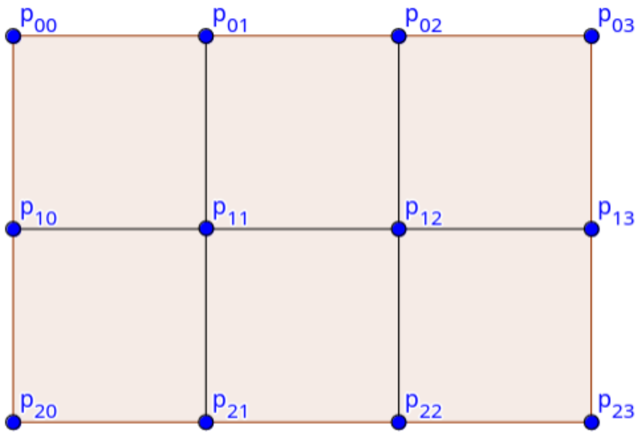


Figure 2: Example of grouping points in rectangles

The triangles generated for all patches are then joined together in a single vector to create the Shape object, which is then dumped to the desired output file. Example of rendering the provided teapot patch is shown in Figure 3.

6 Conclusion

As a result of this third phase, the group has migrated the engine to use VBOs from immediate mode, implemented animations using Catmull-Rom curves, and added more possibilites to generate shapes using Bezier patches. All provided test files yielded the expected result, strengthning the belief that this phase went well. There are always ways to improve: in this instance, we could have created a more complex shape for the commet instead of using the provided teapot; and, like stated before, indexed VBO's could have been used.

Overall, and in conclusion, the group believes this phase went well and is looking forward to the end this project on a high with the fourth and final phase.

References

[1] António Ramires Fernandes. *Curves and Surfaces*. `https://elearning.uminho.pt/bbcswebdav/pid-1310583-dt-content-rid-6643869_1/courses/2223.J306N2_2/curvas\%20e\%20superf\%C3\%ADcies.pdf`. [Accessed: 19-Apr-2023]. 2020.

[2] *Initialization - cppreference.com — en.cppreference.com*. `https://en.cppreference.com/w/cpp/language/initialization`. [Accessed 23-Apr-2023].

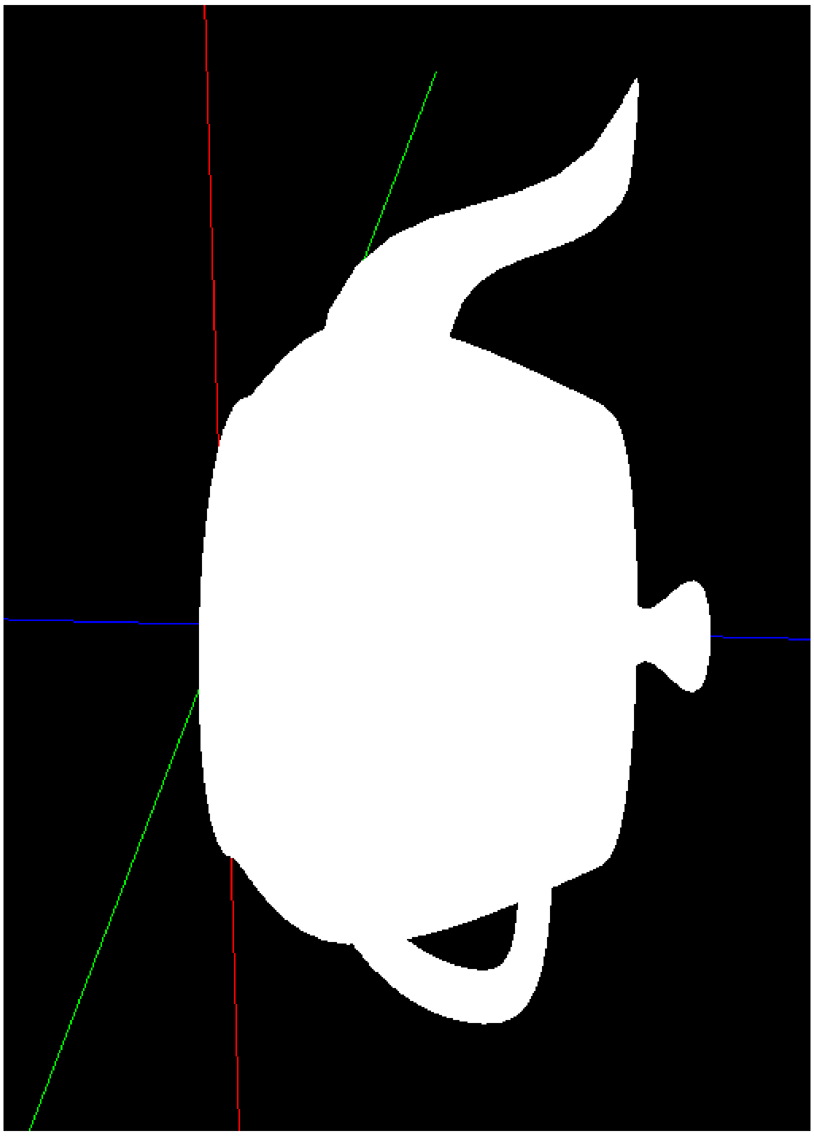


Figure 3: Rendering of the provided teapot