Assignment 2 — Tessellation and Interpolation

Lund University Graphics Group

In this assignment you will tessellate your own model from a parametric equation. This task involves setting up appropriate data structures, and then generating surface points and surface derivatives from the parametric equations — an exercise in programming as well as vector calculus.

You will also implement two different interpolation schemes: linear interpolation and cubic interpolation using Catmull-Rom splines.

1   Tessellation

In this section, we will look at tessellation itself but also how to upload all that data to the GPU and tell it how it should be used, and we will start by looking at the latter with a simple example.

Exercise 1:
We will start by making some modifications in `src ‣ EDAF80 ‣ assignment.cpp`:

1. Change the camera translation to $\begin{pmatrix} 0.0 & 0.0 & 0.5 \end{pmatrix}$, for it to be closer to the geometry being edited. Look for the comment "Set up the camera", and the `SetTranslate` call right after it is the one you want to change.

2. A few lines above the camera translation, replace `createCircleRing(2.0f, 0.75f, 40u, 4u)` with `createQuad(0.25f, 0.15f)`. Note that if you try to start the program now, it will close instantly: this is due to the implementation of `createQuad()` being incomplete.

Finish the implementation of `createQuad()` (found in `src ‣ EDAF80 ‣ parametric_shapes.cpp`) by resolving all the remaining todo items found in that function.

The current implementation does not perform any tesselation; that will be addressed in assignment 4.

You should now be greeted by a white quad on screen (see Figure 1) when running the program. If you want to better see the underlying geometry (i.e. the triangles it is made of), switch in the GUI the polygon mode from "Fill" to "Line" and you should now be able to distinguish the two triangles making up the quad (see Figure 2).
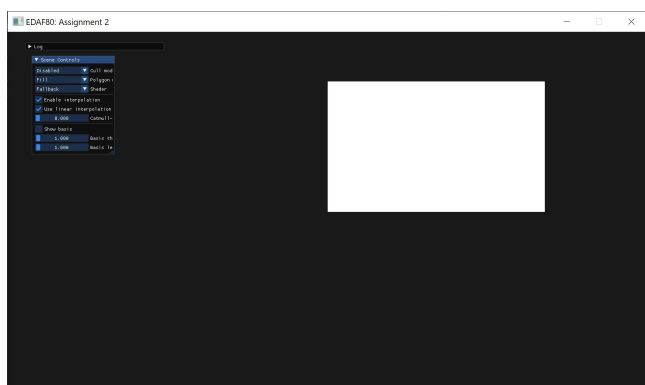

Figure 1: Quad in fill mode.

Next you will be tessellating at least one shape, a sphere in this case. You will still need to also update the data to the GPU and configure its vertex array object, but you should now know how to do so.
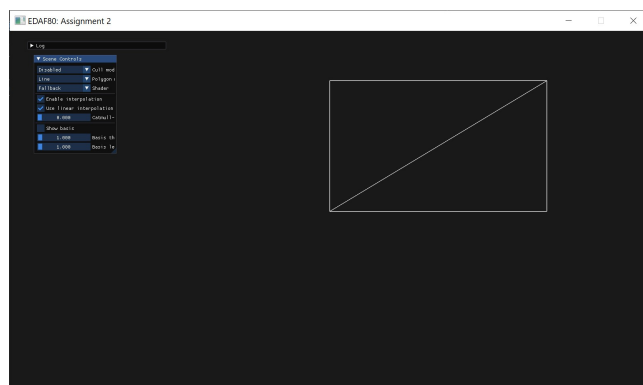

Figure 2: Quad in line mode to highlight the triangles.

Below you can find the different equations for the vertex position (1a), tangent (1b), and binormal (1c) of a sphere. Note that for each of them, the following conditions should hold: $0 \le \theta \le 2\pi$ and $0 \le \phi \le \pi$.

$$\mathbf{p}(\theta, \phi) = \begin{Bmatrix} r\sin(\theta)\sin(\phi) \\ -r\cos(\phi) \\ r\cos(\theta)\sin(\phi) \end{Bmatrix} \tag{1a}$$

$$\frac{\partial \mathbf{p}}{\partial \theta} = \begin{Bmatrix} r\cos(\theta)\sin(\phi) \\ 0 \\ -r\sin(\theta)\sin(\phi) \end{Bmatrix} \tag{1b}$$

$$\frac{\partial \mathbf{p}}{\partial \phi} = \begin{Bmatrix} r\sin(\theta)\cos(\phi) \\ r\sin(\phi) \\ r\cos(\theta)\cos(\phi) \end{Bmatrix} \tag{1c}$$

Exercise 2:
Replace the quad with a sphere created using `createSphere(0.15f, 10u, 10u)`, and comment out the rendering of the control points for now.

Using Equations (1) as well as your knowledge newly acquired in the previous step, implement `createSphere()` (in the same file as `createQuad()`). You will need to

1. generate the various vertex attributes (position, normal, tangent, binormal, and texture coordinates),

2. generate the indices to group the vertices into triangles,

3. upload all that data to the GPU,

4. configure the *vertex array object*.

You can use the existing implementation of the circle ring as guidance; make sure to normalise all your normals, tangents, and binormals.

Note

- texture coordinates will be dealt with in the next assignment;

- do not optimise by having a single shared vertex for the top, and for the bottom of the sphere: this will later cause issues when applying textures. You should instead use the same logic as for all the other slices of your sphere, even if that means having multiple vertices in the exact same location in the end;

- for the slice whose angle goes from 0 to 2π, the last vertex should have the exact same location as the first vertex on that same slice, and they should not be connected by triangles (as those would be too small to be

seen anyway); this is again to prevent issues later on when applying textures.

A few tips while implementing and debugging `createSphere()`:

- Try using a low amount of splits (for example 2 longitudinal splits and 1 latitudinal splits).
- Switch to the "line" polygon mode (accessible through the GUI, in the "Scene controls" window) as it allows you to see the individual edges.
- Enable back-face culling (accessible through the GUI, in the "Scene controls" window) and check that you still see the front of the sphere but the back is gone (see Figure 3); this is easier to do while also using the "line" polygon mode. If the front faces are gone, you are specifying your vertices in clock-wise ordering rather than counter clock-wise, so you will need to tweak your indices.
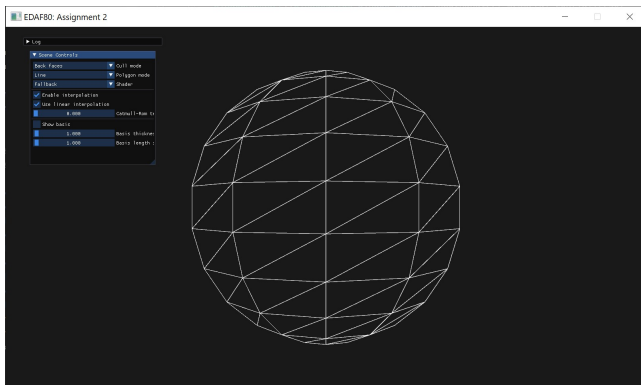

Figure 3: Rendering the sphere with the fallback shader, backface culling enabled, and using the "Line" polygon mode.
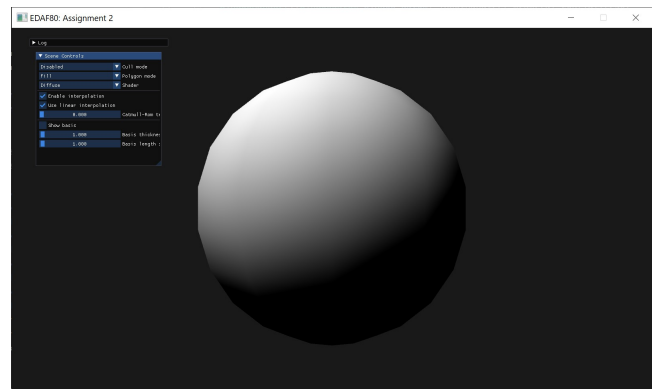
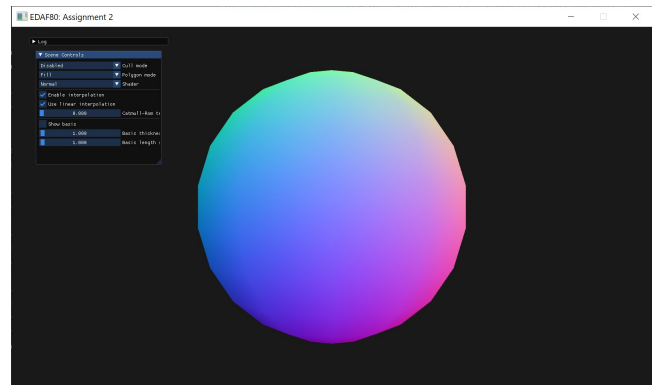- Use the different shaders at your disposition (selectable from the GUI, in the "Scene controls" window) to verify that your normals (Figure 5), tangents (Figure 6), and binormals (Figure 7) are correct.

  In particular, do not forget to simplify the equation of the tangent, to get rid of the black hole at the bottom of the sphere when using the tangent or normal shaders. If you get a match with all the previous screenshots, place the camera at $\begin{pmatrix} 0 & -0.5 & 0 \end{pmatrix}$ and call `mCamera.mWorld.SetRotateX(glm::half_pi<float>());` to position the camera below the sphere and looking up. If you see a black hole similar to Figure 8 when viewing the tangents, then you forgot to simplify the equation for computing the tangent; the expected result is show in Figure 9.
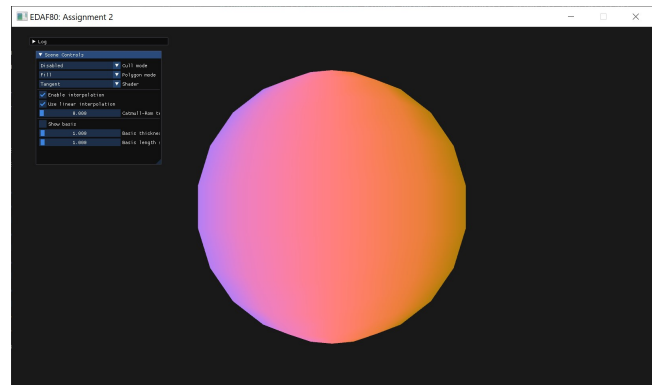
Below you can find the different equations for the vertex position (2a), tangent (2b), and binormal (2c) of a torus. Note that for each of them, the following conditions should hold: $0 \leq \theta \leq 2\pi$ and $0 \leq \phi \leq 2\pi$.

$$\mathbf{p}(\theta, \phi) = \begin{Bmatrix} \big(r_a + r_b \cos(\theta)\big) \cos(\phi) \\ -r_b \sin(\theta) \\ \big(r_a + r_b \cos(\theta)\big) \sin(\phi) \end{Bmatrix} \tag{2a}$$

$$\frac{\partial \mathbf{p}}{\partial \theta} = \begin{Bmatrix} -r_b \sin(\theta) \cos(\phi) \\ -r_b \cos(\theta) \\ -r_b \sin(\theta) \sin(\phi) \end{Bmatrix} \tag{2b}$$

$$\frac{\partial \mathbf{p}}{\partial \phi} = \begin{Bmatrix} -\big(r_a + r_b \cos(\theta)\big) \sin(\phi) \\ 0 \\ \big(r_a + r_b \cos(\theta)\big) \cos(\phi) \end{Bmatrix} \tag{2c}$$

Exercise 3 (Optional):

Figure 4: Sphere front with the diffuse shader.

Figure 5: Sphere front with the normal shader.

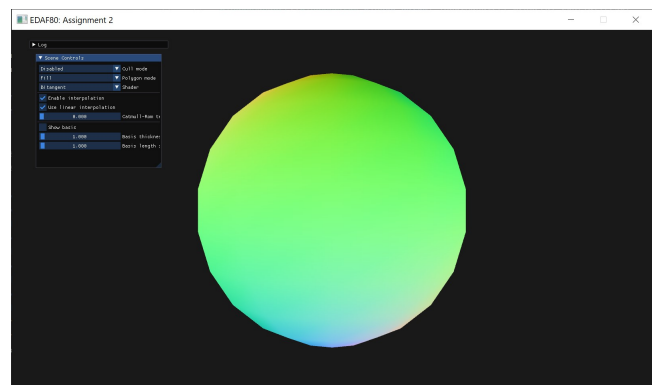Figure 6: Sphere front with the tangent shader.

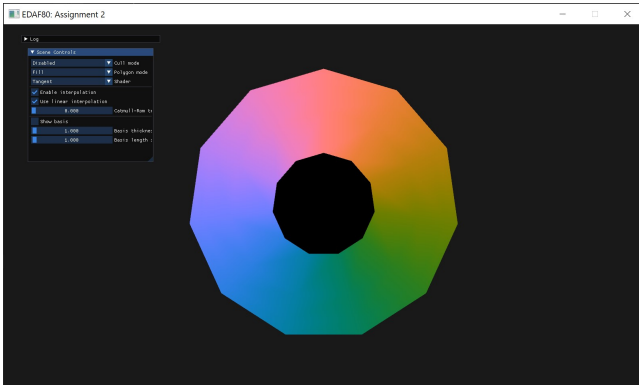Figure 7: Sphere front with the bitangent shader.

Figure 8: Bottom of the sphere with the tangent shader, when the tangent equation *has not* been simplified.
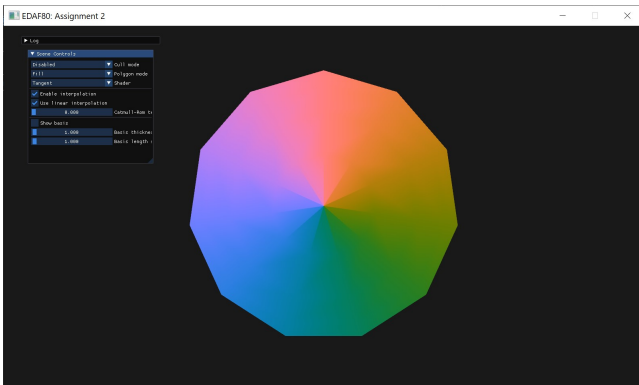

Figure 9: Bottom of the sphere with the tangent shader, when the tangent equation *has* been simplified.

Using Equations (2), implement `createTorus()`. The same recommendations as for `createSphere()` apply here.

2 Interpolation

Linear interpolation:, with $x \in [0, 1]$

$$\mathbf{p}(x) = \begin{bmatrix} 1 & x \end{bmatrix} \begin{bmatrix} 1 & 0 \\ -1 & 1 \end{bmatrix} \begin{bmatrix} \mathbf{p}_i \\ \mathbf{p}_{i+1} \end{bmatrix}$$

Catmull-Rom spline, with $x \in [0, 1]$:

$$\mathbf{q}(x) = \begin{bmatrix} 1 & x & x^2 & x^3 \end{bmatrix} \begin{bmatrix} 0 & 1 & 0 & 0 \\ -\tau & 0 & \tau & 0 \\ 2\tau & \tau - 3 & 3 - 2\tau & -\tau \\ -\tau & 2 - \tau & \tau - 2 & \tau \end{bmatrix} \begin{bmatrix} \mathbf{p}_{i-1} \\ \mathbf{p}_i \\ \mathbf{p}_{i+1} \\ \mathbf{p}_{i+2} \end{bmatrix}$$

The tension factor can be set to $\tau = 0.5$ initially.

Exercise 4:
1. Move the camera back to $\begin{pmatrix} 0 & 1 & 9 \end{pmatrix}$, remove the rotation on the camera, and re-enable the rendering of the control points.

2. Implement the function for linear and cubic Catmull-Rom interpolation in the file `src ▸ EDAF80 ▸ interpolation.cpp`.

   Attention   Bear in mind that matrices in GLM (similarly to GLSL) are using column-major ordering, i.e. the first index gives you a column and the second will give you the corresponding element within a column. So a `glm::mat3x4` specifies a matrix with 3 columns and 4 rows, as opposed to the mathematical notation which would be specifying a matrix with 3 rows and 4 columns.

Table 1: Various controls when running an assignment. "Reload the shaders" is not available in assignments 1 and 2 of EDAF80, while "Toggle fullscreen mode" is missing from assignment 2 of EDAN35.

Action	Shortcut
Move forward	W
Move backward	S
Strafe to the left	A
Strafe to the right	D
Move downward	Q
Move upward	E
"Walk" modifier	⇧
"Sprint" modifier	Ctrl
Reload the shaders	R
Hide the whole UI	F2
Hide the log UI	F3
Toggle fullscreen mode	F11

Similarly when constructing a matrix, you can either give it the different vectors representing the *columns*, or give all the elements individually as floating-point values but make sure to specify them in column-major order.

3. Have an object of your choice interpolate along the path stored in `control_point_locations`. To translate the node to a specified position, use the following code

   `node.get_transform().SetTranslate(newPosition);`

   Make sure to compute the interpolated position within the `if (interpolate) {}` block so that you can pause the interpolation from the GUI, if you need to debug or inspect something. You can use the variable `ellapsed_time_s` to help you with animating your interpolation.

   All the control points are visualised by a small sphere (once you have implemented `createSphere()`) to help you visualise the interpolation path.

3 Discussion Topics

- How does the distribution of the control points affect the velocity of the animated objects? Why? How can this be addressed?

- Animated two objects along the same path using different tension τ; how do the trajectories differ?

- How can an animated object be made to keep "facing forward" as it travels along its path?

  Hint: Use the derivative of the spline (analytical or numerical) to obtain a tangent vector.

A Framework controls

The framework uses standard key bindings for movement, such as W , A , S , and D . But there are also custom key bindings for moving up and down, as well as controlling the UI. All those key bindings are listed in Table 1.

There is only one action currently bound to the mouse, and that is rotating the camera. To do so, move the mouse while holding the left mouse button.

GUI elements can be toggled being a collapsed and expanded state by double clicking on their title bar. And they can be moved around the window by dragging their title bar wherever desired (within the window).

Table 2: Various keyboard shortcuts for Visual Studio 2019 and 2017, and Xcode.

Action	Shortcut	
	Visual Studio	Xcode
Build	`Ctrl`+`B`	`⌘`+`B`
Run (with the debugger)	`F5`	`⌘`+`R`
Run (without the debugger)	`Ctrl`+`F5`	
Toggle breakpoint at current line	`F9`	`⌘`+`\`
Stop debugging	`⇧`+`F5`	`⌘`+`.`
Continue (while in break mode)	`F5`	`ctrl`+`⌘`+`Y`
Step Over (while in break mode)	`F10`	`F6`
Step Into (while in break mode)	`F11`	`F7`
Step Out (while in break mode)	`⇧`+`F11`	`F8`
Comment selection	`Ctrl`+`K`,`Ctrl`+`C`	`⌘`+`/`
Uncomment selection	`Ctrl`+`K`,`Ctrl`+`U`	`⌘`+`/`
Delete entire row	`Ctrl`+`X`	

B IDE key bindings

To help with getting certain tasks done more efficiently, Table 2 lists key bindings of different IDEs for several common actions.