2D To 3D Sketchpad

Visual Computing, Take-Home Exam
Github Repository

Oliver Lemonakis, 201910301@post.au.dk

December 2025

1 Introduction and Motivation

For the last assignment of the Visual Computing course at AU, we are tasked with making a mini-project which bridges the gap between computer vision and computer graphics (Visual Computing). I chose to build a "2D to 3D Sketchpad", which bridges said gap.
My project shows the user a 2D sketchpad window, in which the user is able to draw either a triangle, square, or pentagon contiguous shape in whichever color they specified in the trackbars above the sketching window. Once the user is satisfied with the shapes that they have drawn, they can have the shapes rendered in 3D in the rendering window. The shapes will retain their relative position that they had in the 2D sketchpad, and will be rendered in chosen color from the sketchpad as well. The user can then extrude the shapes however much they want.
To make this project, computer vision techniques for contour and vertex approximation and computer graphics for 3D rendering and extrusion are both required.

2 Implementation

The first considerations made on how to construct a system which could fulfill the specifications required by my project were that of data. I knew I wanted to forego free-hand drawing, and instead focus on simply shapes like triangles, rectangles, pentagons, etc, but I didn't know what a "Shape" meant in the context of my project. The first steps taken for this project were how to model the data. I wanted a data model for the shapes that could contain all necessary data for 3D rendering, regardless of which shape it was. When considering how to build the system, I had originally envisioned an approach which only required a single window, wherein I would switch out the internal context of the window between either OpenCV or OpenGL. However, this proved to be more difficult to implement than previously conceived. Instead I opted for a two window approach, where we have an open Sketcher window, which can then export the shapes of the Sketcher window to the Renderer, which it in turn can render onto the second window.

2.1 System Overview

In order to give a better idea of the system architecture in mind, here is a diagram detailing the most critical system considerations and architecture, albeit simplified.

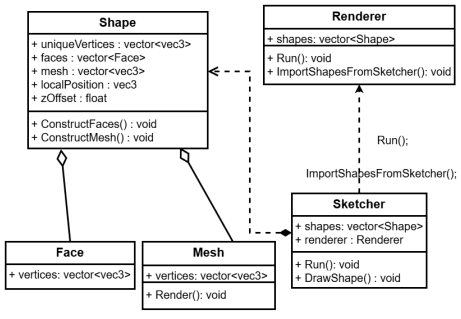


Figure 1: System Architecture, simplified

2.2 Shapes

Shapes are structures that are used to describe both faces of any polygon and mesh of any polygon, with additional information necessary to provide a baseline for the extrusion with a Z-axis offset value and the local position of the shape in the Sketcher.
The fields of the Shape are as follows:

- UniqueVertices
  Self-explanatorily stores the unique vertices of each shape, with respect to all the unique vertices in the shape that we use to construct all the faces of the shape.
- Faces
  Initially, the shape only has a single front-face made of the initial vertices of the face extracted from the sketcher using OpenCV. Upon initialization of the Shape we can construct the rest of the faces by pairing different vertices of the initial face and off-setting them by our z-axis offset value.
- Mesh
  The mesh of our shape, necessary for the rendering of the shape.
- LocalPosition
  Vertex at the center of my shape, describing the local position.
- ZOffset
  An appropriately chosen z-axis offset is needed to build the faces of the shape.

2.3 Meshes

Meshes are what we render in the renderer and are the vertices of our 3D representation. Meshes are constituted of three-dimensional vertices that triangulate the faces of our shape. All of our side faces can be constituted of simple quads, meaning that we only need two triangles, meaning two sets of three vertices for each of the side faces. A generalized formula for the amount of triangles necessary to triangulate a face in a n-sided polygon is N-2. We also calculate and store the triangle normals for properly orient the mesh. Local center is also stored for the mesh, in order to correctly origin the mesh in our renderer.

2.4 2D Sketchpad

The 2D Sketchpad or "Sketcher", is the first thing you're met with when you boot up the application. The Sketcher is where you will be drawing all of the shapes that you wish to render in 3D. You can choose between triangles, squares, and pentagons and choose which color you would like them to be in.

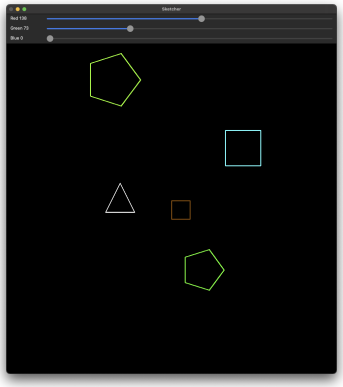


Figure 2: 2D Sketchpad Window

Drawing a shape onto the sketcher triggers the following flow:

1. You choose your shape using keys one-through-three and choose your color using the trackbars in the window. If at any time you are unhappy with what you have drawn, you can clear the canvas by pressing "c".
2. With the left mouse button, we call "DrawPreview" and preview the shape we are drawing by using OpenCV's Rectangle or PolyLine methods. The shapes should be non-contiguous and overlapping shapes might cause unexpected behaviour. Once we let go of the mouse button we permanently draw our shape onto the canvas in whichever size and color we want.
3. Letting go of the triggers the "DrawShape"

method, which will not only draw our finalized shape onto the canvas, but we will also begin to extract our shapes from our canvas.

4. In order to extract the shapes from our canvas we first need to approximate the contours in the canvas, which we do with the OpenCV "FindContours" method. We convert our image into grey-scale and call the method on our canvas. The method will return all contours found in a array, and the number of contours entries in this array will denote the amount of non-contiguous shapes that we have drawn. Once we have our array of contours, we go over each entry in the array and use "ApproxPolyDP" to approximate the most critical vertices of our contour, meaning the corners in our case. These corner vertices are then used to initialize each of our shape objects and we construct the faces from said vertices.

5. Once you are ready to proceed you can press enter/return, which will open the rendering window. The Sketcher orchestrates the exportation of shapes from our 2D canvas to the renderer and prompts the renderer to run.

2.5 3D Renderer

Now that we have our 3D Renderer open, or just renderer, we are ready to proceed.

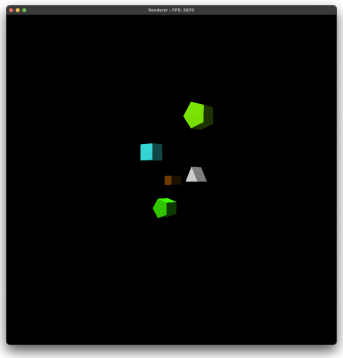


Figure 3: 3D Rendering Window

1. We create our GLFW window and context and load OpenGL.

2. The 3D renderer receives the shapes extracted by the sketcher and subsequently has the mesh for each shape constructed. We do not construct the mesh of the shape at instantiation, as the mesh requires an OpenGL context that is not active at the time of instantiation. This seems reasonable, as there is no need to construct the mesh before-hand since it is only needed for the renderer.

3. As part of the mesh creation we triangulate each face of the shape and store the vertices for each triangle along with the normal for each vertex. Thus using the vertices from each face we can build our VBO and VAO for each shape.

4. We initialize our shader using our vertex and fragment shaders.

5. We need to pass perspective, view and model matrices to our vertex shader in order for us to setup our camera and viewport as we want it to look in the renderer. In this case we have a spinning view of the scene.

6. We also need to light and color the polygons we've extruded and we setup a directional light in our scene, and our fragment shader has some ambient lighting effects we use so that our polygons are never completely dark on either face. We pass the color of each shape off to our fragment shader so we can color our faces accordingly.

7. Lastly, we render the mesh for each shape.

At this point the user can manipulate the extrusion of the shapes in the scene by using the "ArrowUp/Down" keys. This is done by manipulating the z-axis offset value present in the vertex shader. If the user wishes to go back and draw some other shapes, they can do so by pressing the "Esc" key, at which point the rendering window will close and the sketcher will be cleared of shapes.

3 Experiments and Analysis

While I am very confident in the performance of my system, I presume that I can make some hypotheses about what I believe will be flaws or drawbacks of the system.

1. There will be an exponential decrease in extraction performance corresponding to the amount of shapes to extract.
2. The same will also be true for mesh construction, as I believe it will take compoundingly longer time for more shapes to have their mesh constructed.
3. I believe that we will see an exponential decrease in rendering performance corresponding to the amount of shapes drawn on the screen.
4. Not only will more shapes decrease performance, but the choice of shape will also impact performance in both instances.

To test these hypotheses, the following questions arose.

- How does the amount of shapes impact the extraction performance?
- Will the user's choice of shape impact the performance of the system?
- Does the amount of shapes impact the rendering window performance?

3.1 Experiments

In order to fulfill the questions which test my hypotheses, several experiments have been conducted. We run through each shape available to the user, and measure system performance in milliseconds between method calls for extraction performance and mesh construction. Additionally, we measure a rounded average of the rendering performance of our scene, in a ten second rendering interval. The experiments were conducted by manually drawing the shapes inside of the window and subsequently checking the console output for the values or reading the rendering window's FPS count.

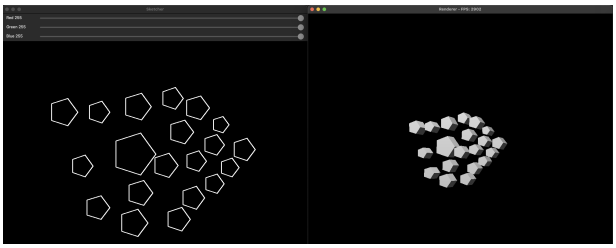


Figure 4: Pentagon Experimentation Example

We draw each shape one through fifty times, while collecting performance data in order to be able to test our hypotheses.

4 Results and Discussion

4.1 Results

Let us start off with the results of the shape extraction and drawing time.

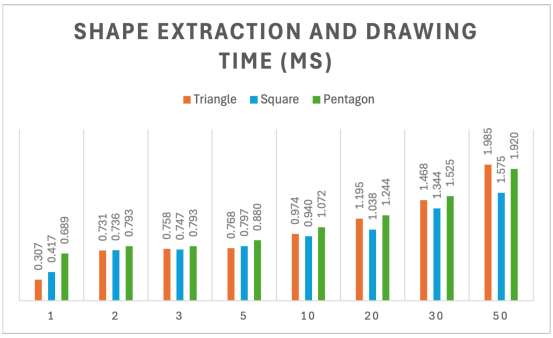


Figure 5: Shape Extraction and Drawing Time Chart

The results1 of the extraction time and drawing time shows us that we do in fact not see a exponentially higher decrease in system performance when it comes to our systems ability to find the contours of each shape and subsequently approximating the vertices and building the faces. The results hint to us that there is instead a linear relation between the amount of shapes in our 2D Sketchpad and the amount of time it takes us to draw, extract, and builds the shapes. For the initial shape we do see quite a large

difference between the systems performance of the different shapes. However, when we get to ten to fifty shapes both triangles and pentagons are less performant than squares, with triangles being the most intensive of our shapes in the last test of fifty non-contiguous shapes.
Next, the mesh construction time results.

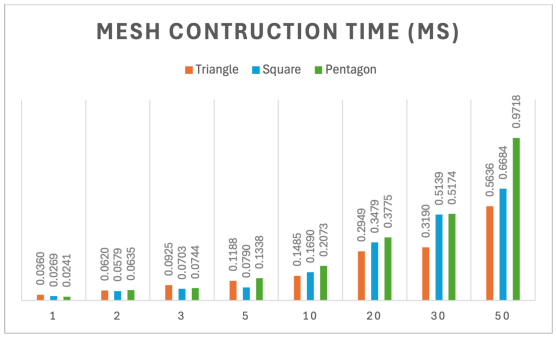


Figure 6: Mesh Construction Time Chart

For our results of mesh construction time we don't see an exponential growth in the time it takes for the system to build the meshes for the shapes. Across all amounts of shapes2 the triangles are the least intensive shapes to construct meshes for, with the squares following and the pentagons being the most intensive. And lastly, the rendering window's performance, measured in frames-per-second.

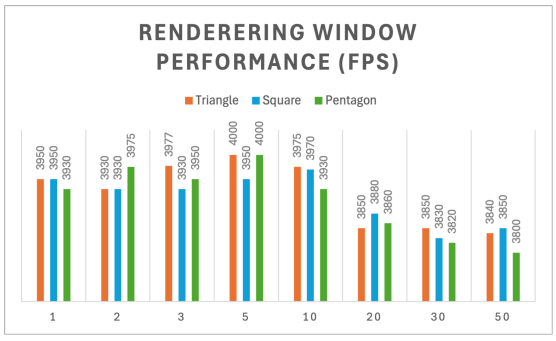


Figure 7: Rendering Window Performance Chart

While initially there does not seem to be any amount of difference between how many shapes we have in the scene and performance in FPS in the rendering window, we see an decrease in FPS as we scale up past 10 shapes rendered in the scene. While we have large fluctuations in values in the amounts of shapes ten and below, we still see pentagons being the most strenuous for the system to render after fifty shapes are introduced in the scene3.

4.2 Discussion

What does the results mean for my hypotheses?
Let's start off with the first hypothesis and the results gathered from the experiment. Is shape extraction performance impacted exponentially by the amount of shapes drawn on the canvas? The results say no, they are not exponentially impacted, but linearly impacted. What does this tell us? It tells us, that while I believe it would become increasingly harder and harder for my system to perform the task, it did not in fact. Why could this be? Let's start off with the drawing. No matter how many shapes exist in my system, my "DrawShape" method which is in charge of drawing the final shape to the canvas, but also extracting and building the shapes off of the canvas, only ever draws one shape. We can safely expect it to be equally straining on the system's performance no matter how many shapes already exist on the canvas. But can we say the same thing for the shape extraction?

When we do our shape extraction, we clear the vector of all previous shapes, and we use the OpenCV "FindContours" method to find all contours arrays of the shapes in the canvas. When we conduct the experiment, and we go over the canvas for five, ten, thirty shapes, the amount of contour approximation we do should actually be mostly the same. There can be differences between the amount of vertices in our contours, but when we approximate the most critical points of the we limit ourselves to the exact same amount of vertices for each consecutive shape. Therefore, it does actually make sense that we don't see an exponential growth in computational complexity between ten and thirty shapes, but a linear one instead.

For the mesh construction results, this line of think-

ing tracks as well. If we have twice the amount of meshes to construct, we would expect the amount of time needed to construct said meshes to be doubled, not triples or quadrupled. The results show just that. Even when considering the baseline we see an almost perfectly linear trend between the amount of shapes and the time it takes for us to construct the mesh for said shape. We also see that more complex polygons take longer to construct than less complex ones. This makes sense again, because we do have more triangles to construct. Since the amount of triangles a n-sided polygon consists of is N-2, we definitely expect the pentagons to be the most expensive to construct. But we see that it's only marginal, and not twice the amount of construction time between triangles and squares for example. This is explained by the fact that, while we have can have a more or less complex front-face and back-face, the side-faces which constitute most of the shapes, will always be square and require two triangles.

The rendering results tells us that there is little to no difference between the performance before we go beyond ten shapes. Once we are rendering 20 shapes, the performance drops ever so slightly, but realistically its only a performance decrease of around three to four percent, and the drop between twenty and fifty shapes is lower still. We would most likely need to construct a test environment, wherein we can try rendering thousands of shapes, meaning tens of thousands of primitives, instead of hundreds as we do right now. At the amount of shapes we are testing it is difficult to argue that the results of the drop in rendering performance are anything but speculative at best.

The fourth and last hypothesis, which concerns the performance of different shapes, fared better than the others. Some outlier values notwithstanding, the hypothesis that we would see an increase in the computation needed to extract and construct shapes, construct meshes, and render those meshes depending on which shape the user chose was true. This was true across all three experiments.

Now, why do my hypotheses expect me to see a exponential decrease in performance with respect to the amount of shapes both drawn and rendered in the application? When I had done my initial pre-experiment testing, I discovered a bug in my program. Every time we drew a shape, we went over the canvas and added all the shapes we discovered with out shape extraction method to our list of shapes. This meant that if we had five shapes in our canvas and drew a sixth, we would add the six shapes in our canvas to our existing list of five shapes, which in turn would not have five shapes in it but fifteen. This means that I kept a running total of shapes, instead of the total of shapes in the canvas. The results of my initial experiments are available in the appendix section8, and shows the results caused by this bugs unintended behaviour.

5 Conclusion

My initial assumption was that the performance decrease would be either super-linearly or exponentially decreasing when we introduced more and more shapes into the sketcher. During development and initial testing I saw a development towards such a conclusion which led me to preemptively hypothesize what results I see. Contrary to my hypotheses the system performance decreased linearly rather than exponentially. I did, however, observe differences in performance between the different shapes rendered using my 2D to 3D Sketchpad.

6 Appendix

6.1 Data

T, S, P denotes triangle, square, and pentagon respectively. The number above the results are the amount of said shapes present in both sketcher and renderer.

0.307	0.731	0.758	0.768	0.974	1.195	1.468	1.985
0.417	0.736	0.747	0.797	0.940	1.038	1.344	1.575
0.689	0.793	0.793	0.880	1.072	1.244	1.525	1.920

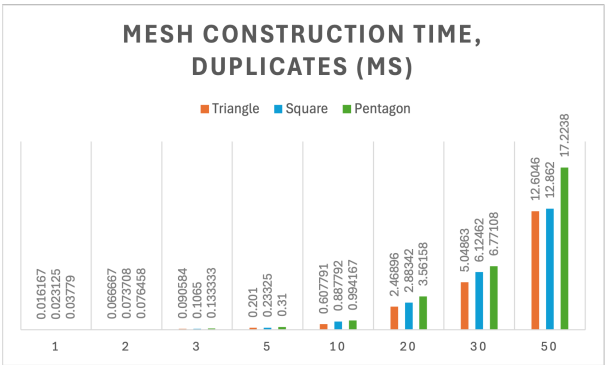
Table 1: Shape Extraction and Drawing Time (ms)

	1	2	3	5	10	20	30	50
T	0.0360	0.0620	0.0925	0.1188	0.1485	0.2949	0.3190	0.5636
S	0.0269	0.0579	0.0703	0.0790	0.1690	0.3479	0.5139	0.6684
P	0.0241	0.0635	0.0744	0.1338	0.2073	0.3775	0.5174	0.9718

Table 2: Mesh Construction Time (ms)

Type of Shape	1	2	3	5	10	20	30	50
Triangle	3950	3930	3977	4000	3975	3850	3850	3840
Square	3950	3930	3930	3950	3970	3880	3830	3850
Pentagon	3930	3975	3950	4000	3930	3860	3820	3800

Table 3: Rendering Window Performance (FPS)



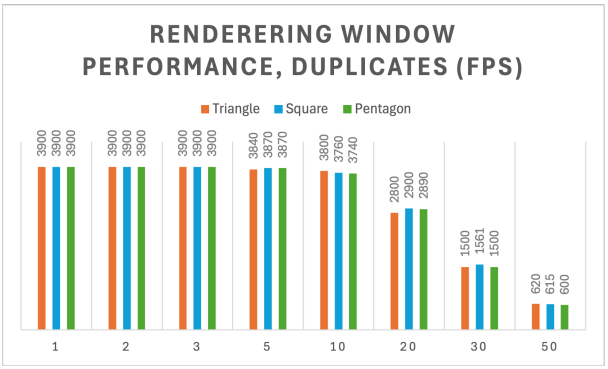


Figure 8: Duplicate Shapes Charts

	1	2	3	5	10	20	30	50
T	0.0162	0.0667	0.0906	0.201	0.6078	2.4690	5.0486	12.6046
S	0.02313	0.07370	0.1065	0.2333	0.8878	2.8834	6.1246	12.862
P	0.03779	0.07646	0.1333	0.31	0.9942	3.5616	6.7711	17.2238

Table 4: Mesh Construction Time, Duplicate Shapes (ms)

	1	2	3	5	10	20	30	50
T	3900	3900	3900	3840	3800	2800	1500	620
S	3900	3900	3900	3870	3760	2900	1561	615
P	3900	3900	3900	3870	3740	2890	1500	600

Table 5: Renderer Performance, Duplicate Shapes (FPS)