Adding 3D to TSGL

Senior Project Report

Ian Adams

Calvin University

Spring 2020

Project Vision and Overview

The initial vision for this project was to expand the functionality of Calvin University's Thread Safe Graphics Library (TSGL) such that it would have three-dimensional graphical capabilities and to improve TSGL's installation process on Linux, MacOS, and Windows. The project began with a working object-oriented version of TSGL that ran on Linux and MacOS, utilizing underlying OpenGL calls. But in fall 2019, TSGL was limited to two-dimensional functionality. In order to provide a more complete parallel graphical experience, increased utility was an important next step for TSGL. Perpetuating logical progression for TSGL's functionality, three-dimensional graphical capabilities were added to the library while maintaining its thread-safe status for this senior project. In the process, some preexisting functionality was temporarily lost. While much of what was lost has been restored, extra restoration will be required before the current version has the required functionality to be considered stable. As such, the work done has not yet been consolidated into the master branch of the TSGL GitHub repository. The current version of the project that contains a large amount of the desired functionality resides within the SeniorProjectIA branch. With modification and improvement the branch will ideally be ready to merge with master by the end of summer 2020.

Background

TSGL is Calvin's Thread-Safe Graphics Library, the world's only graphics library with thread-safe graphical capabilities. Put succinctly, each thread can add objects to be drawn to the screen to a queue in a single Canvas class, resulting in multithreaded control over the

graphical display. TSGL has been in development since mid-2014, having been worked on by a variety of students including Patrick Crain, Mark Vander Stel, Chris Dilley, Christiaan Hazlett, Elizabeth Koning, and Ian Adams. Since it is built on top of OpenGL, TSGL can be ported to any operating system on which OpenGL calls are supported. This includes Linux, MacOS, and Windows. Over the course of summer 2019, Ian Adams expanded TSGL's functionality to make it object-oriented with the intention of running it on a Raspberry Pi. However, over the course of this summer research only two-dimensional (2D) object-oriented capabilities were added; TSGL could draw Squares, Triangles, and flat Images to the screen, but couldn't easily draw Cubes, Pyramids, or other three-dimensional (3D) objects. Additionally, while TSGL contained install scripts for Linux, MacOS, and Windows, they were somewhat lacking; error-prone and messy, these remained imperfect at the end of the summer and required improvements.

System Design

The first step towards the completion of this project was to decide precisely what 3D objects the improved library should be able to draw. Two object characteristics were vital: objects with enough flexibility that the user didn't feel unnecessarily restricted, but also with relative simplicity such that the user wouldn't feel overwhelmed by the number of parameters or ways to mutate their object. In order to decide which objects matched these criteria, various 3D modeling tools like Blender [6] were examined to see what sort of functionality they provided their users. The theory was that whatever 3D objects a modeling tool allowed its users to easily insert, a basic 3D graphical library should allow its users to utilize as well. After a satisfactory amount of research into 3D modeling software and basic 3D geometrical shapes, it was decided that eight objects would have their own classes to make up the foundation of

TSGL's 3D graphical capabilities: Prism, Cuboid, Cube, Cylinder, Pyramid, Cone, Ellipsoid, and Sphere. These classes were originally placed in a class inheritance structure beneath the Object3D class, which in turn inherited from the Drawable class (from which TSGL's 2D classes also inherited). However, by the end of the project's lifespan, the Object3D class' functionality had been merged into the Drawable class, meaning that all of the 3D classes inherited from Drawable either directly, without a mediating superclass, or indirectly, with one of the other 3D classes as the mediating superclass.

The classes were designed to have relatively few mutable parameters. A Prism was decided to be a multi-sided object with a certain "radius" and height. Essentially, viewed from the top, it appeared to be a regular polygon with a parameterized number of sides. Each vertex on the polygon was to be the specified radius away from the center of the polygon. It was planned that the user would be able to mutate both the radius of the Prism and its height. The possibility of allowing the user to pass in arbitrary vertices through an array and constructing a Prism from them was contemplated, but eventually rejected as there would be no way to ensure that all parameter vertices would be within a single plane in 3D space.

The Cylinder was very similar; it was designed to be a circular shape with a parameterized radius, extended through space to be a parameterized height. Colloquially, it was like a pipe with capped ends. Since like Prism its only two mutable parameters were its cylindrical radius and height, it made sense to have it directly inherit from the Prism class. A Cylinder could be considered a Prism with a great many sides, to the point where the n-sided regular polygon that forms the basis for the Prism is indiscernible from a circle.

Most simply expressed, the Cuboid was designed to be the 3D version of a rectangle. A rectangle is a four-sided 2D object with four right angles between edges, and two pairs of edges that match in length. A Cuboid, on the other hand, is a six-sided 3D object where each side is a rectangle that matches the opposite rectangle. Its width, height, and depth are not required to be equal, and additionally were intended to be separately mutable.

The Cube was designed to be a more restricted version of the Cuboid. While a Cuboid allows for sides of different lengths and contains three pairs of matching rectangles, a Cube requires each side to be a square that matches all five other squares. Instead of having potentially different width, height, and depth, it merely has a single parameterized side length which it utilizes for all three dimensions in 3D space. The possibility of having the Cube inherit from the Cuboid was contemplated, but eventually rejected, as it wasn't logical to have a Cube with the inherited ability to have its width, height, and depth mutated individually.

A Pyramid was to be somewhat similar to a Prism. Taking identical parameters in sides, radius, and height, it was designed to have a regular polygon base with a parameterized number of sides. Each vertex in this base was also the parameter radius away from the center of the regular polygon. However, rather than the edges extending at right angles from the base like a Prism, all edges converged to a single point at the parameterized height above the center of the base, the tip of the Pyramid. The radius and height of the Pyramid were mutable.

A Cone was to the Pyramid as the Cylinder was to the Prism. Similar to a Pyramid, it had a height and radius, but had a great many sides. This made the base indiscernible from a circle, despite being a regular polygon. Cone was a natural subclass for Pyramid, inheriting its radius and height mutability.

The final two classes were Ellipsoid and Sphere. These classes were somewhat similar in design to Cuboid and Cube, respectively. However, instead of being essentially 3D versions of rectangles and squares, Ellipsoid and Sphere were 3D versions of ellipses and circles. An ellipsoid is a round 3D body with potentially different, separately mutable width, height, and depth, much like an ellipse is a round 2D body with potentially different, separately mutable width and height. A Sphere is a more constrained Ellipsoid; it only has a single radius parameter, which it uses to determine its proportions in all three dimensions. Much like Cube was viewed as a potential subclass of Cuboid for a while, Sphere was viewed as a potential subclass of Ellipsoid for a similar duration. However, this idea was eventually rejected as well due to Ellipsoid containing mutators that weren't reasonable for Sphere.

The only other element as far as design went was that each class would have two constructors. The first would take a single ColorGLfloat, a structure that represents a simple RGBA color model with four GLfloat values between 0.0 and 1.0, with each value corresponding to R,G,B, and A, respectively. This single ColorGLfloat would be applied to every added vertex, creating a monocolored 3D object. The second constructor would take an array of ColorGLfloats, with each ColorGLfloat being applied to one or more added vertices depending on the class. This could potentially result in a multicolored 3D object, granting the user more control over the coloration of their object.

Implementation

The first thing implemented was a basic skeleton of the class structure for the new, added 3D functionality. This bare-bones class structure would be the foundation on which the

overall required functionality could be added. It wasn't completely clear what extra private variables would be needed in the final implementation, but in the beginning all known mutable variables were added. Since all 3D object classes initially inherited from the now-removed Object3D class, it contained mutable instance variables such as the center z-coordinate and the rotation point z-coordinate of its inheriting objects, while inheriting the corresponding x and y coordinates from its superclass Drawable, which was initially designed for only 2D objects. Instance variables for yaw, pitch, and roll were also added within Object3D. Basic constructors were defined in all subclasses, and while there was essentially no actual functionality, they defined instance variables and called their corresponding superclass constructor. But no actual vertices were added to be drawn to the Canvas. Most mutators and accessors from the final implementation were given skeleton method declarations, and then either had empty contents or were commented out entirely.

Step two was to use a Cube as an extremely basic 3D example that could be rendered to the Canvas. The primary objective was to update the Canvas class so that adding 3D objects to its queue of objects to be drawn to the Canvas would actually result in them being correctly rendered to the screen. The first decision that needed to be made here was what library to use to create OpenGL contexts and draw to the screen. In TSGL's previous implementation, it used the Graphics Library Framework (GLFW) to create OpenGL contexts, but for a brief while using the OpenGL Utility Toolkit (GLUT) instead was contemplated. GLUT was significantly simpler than GLFW, and additionally, all of the examples within the provided textbook [5] used GLUT. However, it eventually was discovered that GLFW was essentially a straight upgrade to GLUT in terms of power and utility, despite the cost of increased programming complexity. Considering

that at some point, GLFW would have to replace GLUT as the more advanced version of the library anyway, it was decided that the efficient decision would be to find a way to make GLFW continue to work despite its greater complexity.

Updating the Canvas class was important, but first an extremely basic test was needed to make sure that the updates were working to make rendering 3D objects possible. So, the addVertex() method in Object3D was updated to take an extra parameter for the z-coordinate of every vertex, which was previously unneeded considering that the Canvas was strictly 2D. Within the constructor of Cube(), the addVertex() method was called 24 times; one time for each of the four corners of all six sides. The geometryType was GL_QUADS, indicating that every four calls to addVertex() would be used to represent and render a single side. Initially, the vertices entirely ignored all of the parameters in the constructor; given that this minor update to the Cube class was only to provide a way to test if the Canvas class was rendering 3D objects correctly, the Cube class didn't need to care about utilizing its parameters at this point. Instead, Cube just drew a cube of side length one, regardless of parameter values [1]. Object3D's draw() method, which was inherited by all subclasses, was also updated so that once Canvas was appropriately changed, calling draw() on any instance of Object3D or any of its subclasses with vertices appropriately added would result in the instance's initialized vertices being drawn to the Canvas.

Once the Cube class and Object3D class were updated to always add vertices for an extremely basic cube, the Canvas class had to be seriously modified. The main updates made were to alter a few of OpenGL's settings, such as re-enabling the previously unnecessary depth buffer, and entirely revamping the Canvas::draw() method, which again would call

Drawable::draw() on all Drawable objects that had been added to the Canvas, but now with GL_VERTEX_ARRAY and GL_COLOR_ARRAY enabled. This was because the vertices and colors were now stored within two separate arrays rather than one. The other large change that needed to be made to the class was to more or less comment out anything related to OpenGL texturing. While texturing was something that would be necessary to add back in later in order to draw images and font-styled text, the only goal at this point was for the TSGL Canvas to draw a less complicated, untextured graphic, and the necessary OpenGL calls for texturing were creating difficulties with the updated Canvas. In the interests of making progress, these texturing calls were temporarily removed. And progress was made; the vertices added in the Cube were able to render to the screen, and a basic 3D graphic had been drawn. At this point, a minor snag was encountered. For some reason the screen could draw the Cube once, but wouldn't refresh if the vertices were dynamically altered. This, however, was a simple problem of failing to make the current OpenGL context correspond to the correct Canvas every draw cycle, and was easily fixed with a single line of code once the issue had been identified.

Once the Canvas had been updated so that appropriately initialized vertex and color arrays could be drawn to the screen, it's important to note that TSGL couldn't draw anything besides the single extremely basic Cube that had already been updated to work with the new style of drawing. All work beyond this point was more or less purely additive or restorative, depending on whether TSGL was gaining functionality for the first time or whether it was regaining previously possessed functionality.

The next step, now that it had been confirmed that it was possible to draw 3D objects to the Canvas, was to add in more functionality to the Object3D skeleton class structure so that

beyond being able to draw a simple Cube with side length one, it would become possible to draw any or all of the 3D objects in said class structure with expanded functionality in terms of parameterization and mutation. The obvious place to start was with Cube. The first change was to make the scale of the Cube correspond to the side length parameter. From there, the draw cycle was quite simple; rotate the Cube, centered on the origin, around the origin by roll, pitch, and then yaw, and finally translate it to the parameter x, y, and z-coordinates. Mutator functionality was added so that the side length of the Cube could be either set to a new parameter value or altered by a parameter value, with an invariant that the new side length be positive. The side length was altered by modifying the values within the vertices array. With these changes, the Cube functionality was more or less complete.

With one subclass of Object3D complete, the crucial next goal was getting more subclasses to the same level of functionality. The obvious next class to work with was Cuboid. Functionally just a Cube with more flexibility, adding vertices to Cuboid was incredibly easy due to the process just being a slightly altered version of the way they were added in Cube. Additionally, designing mutators that modified the width, height, and depth of Cuboid was also simple. Instead of modifying x, y, and z vertices whenever a side length mutator was called, it was as simple as modifying whichever one of the three corresponded to the width, depth, or height mutator being called, and leaving the other two sets of vertices alone. And with these additions, Cuboid was complete and drawable to the Canvas.

However, now that the two easiest classes had been added, more complicated classes awaited. The greatest challenge with the other classes was deciding in what order to add vertices in coordination with choosing an OpenGL geometry type, with the goal being to most

efficiently render the 3D object correctly. To summarize the final decisions made, a table has been inserted below.

Subclass of Object3D	OpenGL Geometry Type	Description of how vertices are added
Cube	GL_QUADS (every four vertices form a shape to be drawn to the screen)	Four vertices for all six sides; each vertex half of side length away from the origin on the x, y, and z axes.
Cuboid	GL_QUADS (every four vertices form a shape to be drawn to the screen)	Four vertices for all six sides; each vertex half of width, length, and depth on the corresponding axis away from the origin.
Prism	GL_TRIANGLES (every three vertices form a triangle to be drawn)	Twelve vertices per side. A triangle on the top of the Prism, two triangles on the side, and a triangle on the bottom of the Prism.
Cylinder	GL_TRIANGLES (inherited from Prism)	Exact same way as Prism, just with more sides. Inherited functionality.
Pyramid	GL_TRIANGLES (every three vertices form a triangle to be drawn)	Six vertices per side. A triangle from the base edge to the tip, and a triangle from the base edge to the center of the base.
Cone	GL_TRIANGLES (inherited from Pyramid)	Exact same way as Pyramid, with more sides. Inherited functionality.
Ellipsoid	GL_TRIANGLE_STRIP (triangles drawn from vertices (1,2,3), (2,3,4), (3,4,5), etc.	Nested for loops. Vertical strips drawn one at a time, with two vertices added per horizontal strip per vertical strip. Vertices added based on x, y, and z-radii [3].
Sphere	GL_TRIANGLE_STRIP (triangle drawn from vertices (1,2,3), (2,3,4), (3,4,5), etc.	Nested for loops. Vertical strips drawn one at a time, with two vertices added per horizontal strip per vertical strip. Vertices added based on single radius parameter.

One issue that became apparent was the inefficiency of mutating the scale of various objects by manually manipulating the vertices every time the mutator was called. This was enormously evident in Sphere and Ellipsoid. With 1440 vertices each, manually modifying all 1440 x, y, and z values each time Sphere's radius was mutated was foolishly intensive. So, a more efficient solution to this problem was found. OpenGL provides a method that allows for x, y, and z scaling of a drawn object called glScalef(x,y,z) [4]. By creating three new GLfloat

instance variables in the Object3D class representing the x, y, and z scales of the 3D object, and then calling glScalef(myXScale, myYScale, myZScale) before rotation and translation in Object3D::draw(), the subclass mutators and constructors could be significantly simplified.

First, the vertices array no longer had to take into account the constructor arguments, but rather could simply construct a unit version of whatever object was being created. Then, the myXScale, myYScale, and myZScale values would simply be defined by the appropriate parameters, and the unit object would be scaled according to these values in Object3D::draw(). Second, the mutators no longer had to loop through every single vertex in order to alter them. Instead, they could simply modify the appropriate my_Scale instance variable, and through a simple matrix multiplication in glScalef() the modification would be reflected. Instead of controlling scale through the vertices themselves, scale was entirely controlled through these three instance variables. Just as the yaw, pitch, roll, center x-coordinate, center y-coordinate, and center z-coordinate were controlled by six respective instance variables.

The next thing to add was improved rotation capability. While all objects were currently able to rotate around their centers as necessary, it was conceivable that a user might want to rotate their object around an arbitrary point that they could define. This was easily handled by utilizing the three instance variables defining the rotation point x, y, and z-coordinates, and slightly altering Object3D::draw(). Before, the object would simply scale by myXScale, myYScale, myZScale, then rotate by myCurrentRoll, myCurrentPitch, myCurrentYaw, and then translate by myCenterX, myCenterY, and myCenterZ. After the changes, the object would scale, translate to myCenterX, myCenterY, myCenterZ, translate again by negative myRotationPointX, myRotationPointY, myRotationPointZ, rotate appropriately, and translate back by positive

myRotationPointX, myRotationPointY, myRotationPointZ. This functionally handled 3D translation around an arbitrary point [9].

At this point, it became noticeable that monocolored 3D objects weren't particularly discernable as being 3D. A red Sphere, for instance, looked no different than a red circle on the screen. This was because there was no different shading on the objects and no light source in the scene to provide any shadows. Three ideas were proposed to fix this, two of which were implemented. The first proposal was to add a light source to the scene, but this seemed needlessly complicated beyond what the typical user would need, so this was not implemented.

The second proposal was to add outlines to objects making it clear where the edges were. This was relatively easy to implement. While it required very specific slight alterations in the order that vertices were added so that not every edge was outlined and inheriting classes such as Cone and Cylinder wouldn't have the exact same edges outlined as their parents Pyramid and Prism, eventually the problem was solved and the appropriate edges were outlined. In essence, the vertices were drawn a second time, but with the OpenGL geometry type GL_LINES or GL_LINE_LOOP rather than their respective typical geometry type.

The third proposal was that for objects without edges, the coloration should be scaled more or less like a gradient, so that it was darker in certain sections than others. This was also relatively easy to implement. Sphere, Ellipsoid, Cone, and Pyramid all had their monocolor constructors modified so that they would be evidently 3D even without edges displayed.

Another bug was that when an instance of Object3D was rotated around an arbitrary point, its accessed center x, y, and z-coordinates wouldn't change at all despite the fact that when rendered on the Canvas, it would appear to be in a completely different place than the

location stated by the accessors. A solution to this issue was achieved by hand-calculating the result of the matrix multiplications in Object3D::draw() and finding the true center x, y, and z-coordinates after these matrix multiplications had been applied to the vertices [7]. From there, it was as simple as calculating and returning the true coordinate from the appropriate accessor whenever the rotation point wasn't equal to the center of the object and yaw, pitch, or roll was non-zero. If the rotation point was equal to the center of the object or yaw, pitch, and roll were zero, then the unmodified appropriate coordinate would be returned.

At this point, the 3D functionality was considered more or less complete. While the functionality wasn't flawless, it was satisfactory, and attention was turned to restorative work. Since updating the Canvas class to allow for the drawing of 3D objects had made the implementation of all 2D objects no longer viable, alterations were in order to restore the 2D classes to a point where they could be once again be drawn to the Canvas. This involved three things.

First, major changes were made to the way that the class structure looked overall. The most pertinent of these changes was that Object3D's functionality was entirely shifted into the top-level Drawable superclass, and the Object3D class itself was removed. Therefore, all previous direct subclasses of Object3D now inherited directly from Drawable, but overall didn't lose or gain any functionality simply because Drawable had been their indirect superclass previously and all of Object3D's functionality had been kept alive in Drawable. However, for all the 2D classes, this was a massive change. Suddenly, they had all of the functionality that Object3D had. As a result, many classes, such as Shape, had a great deal of functionality that

was no longer relevant removed. The Polygon class was made entirely irrelevant by this change and was deleted in entirety.

From there, the second change was made. All the 2D subclasses, which now had Object3D's functionality but no ability to utilize it, had to be updated so that they were able to be inserted into the 3D space of the Canvas. In short, it was necessary to change the way 2D classes added vertices to their vertex arrays and add all the relevant parameters to the constructors so that the vertices could be inserted into the Canvas' 3D space at a parameterized x, y, and z with a parameterized yaw, pitch, and roll.

The third change was that massive amounts of code had to be commented out. Before the Canvas class was updated to allow for 3D object rendering, TSGL had the functionality to procedurally draw objects to the Canvas a single time and then delete the object, with the object remaining drawn on the Canvas. This was ideal for adding objects that didn't need to have a single aspect of them mutated, but still needed to be drawn to the Canvas. However, in a 3D space it's not so simple to draw something a single time, simply because depth matters in a way it doesn't on a 2D Canvas. With a depth buffer, it's important to discern whether an object is being drawn behind or in front of another, and without memory access to every object on the Canvas, it's impossible to do this. And objects that have been drawn a single time and discarded aren't stored in memory. So, all of the procedural functionality that TSGL previously contained needed to be removed, but in such a way that if a clever individual could find a way to re-add some sort of procedural functionality in the future, all of the ground work would already be there for them and merely in need of uncommenting and modification.

But in any case, almost all 2D object-oriented functionality was restored in a single massive commit, revamping the class structure of TSGL along the way. But a final bug still needed ironing out. The ConcavePolygon class was having issues with the color mutator. This is because after adding the vertices to this class, an extremely convoluted preprocess() method was being called to manually alter the order and number of vertices so that OpenGL could handle them. OpenGL doesn't work with concave polygons easily, simply because it depends on drawing triangles, and with a concave polygon there's no guarantee that a drawn triangle between three vertices won't overlap an edge. However, because this preprocess method resulted in the vertices array being completely reordered and containing an unintuitive number of entries, there was no simple color mutator that would give the same color scheme as the constructor without recreating the content of the preprocess method within the mutator.

This problem was solved by using OpenGL's stencil buffer [2]. Essentially, this allows OpenGL to draw vertices like a normal convex polygon would, but instead of rendering the entirety of every drawn triangle, it only renders the parts within the specified concave vertices. Despite visual indicators that it wasn't perfect, it was enormously simpler, more efficient, and more aesthetically pleasing than the result of the bizarre preprocess method ever was.

Testing

In order to test each added 3D object-oriented class, files called test<class name>.cpp were created and added to the tests folder in TSGL and to the list of test files to compile in TSGL's Makefile. Each of these test files looked relatively similar. First, each file would define a global testing function where all the actual testing would occur, which accepted a Canvas object

as a parameter. The file would then define a Canvas in main() and tell the new Canvas to run() the global test method. Within the test method, the corresponding class that was being tested would have one or more instances of it created and added to the Canvas.

From there, mutators and accessors would be called on the instance(s) of the class that had been added in order to ensure that all method functionality was working as intended. Additionally, on top of the class' mutators and accessors being called to ensure that they were working correctly, all superclass mutators and accessors would be called in order to ensure that inheritance was operating smoothly. The core methods that were tested for essentially every class were mutators of yaw, pitch, and roll, mutators of the center x-coordinate, y-coordinate, and z-coordinate, and mutators of the object's coloring. This is because these were initially contained within the Object3D class, from which all 3D object classes inherited, and every class either would inherit these methods or override them if different functionality was desired. Color mutation was the most frequently overridden, due to differences in how many vertices corresponded to each parameter color. Otherwise, class-specific mutators that altered various corresponding dimensions of the object were similarly tested. An example of this would be mutators for the x, y, and z-radii of the Ellipsoid class being tested in testEllipsoid.cpp.

Similar test methods were added for the 2D objects when their functionality was updated in such a way that TSGL could draw them again. Since the Object3D class was merged into the Drawable class, from which all 2D objects inherited, mutators for the x, y, and z-coordinates, yaw, pitch, roll, and coloring of the 2D objects were tested in these files, just as they were in the 3D test files. Similarly, any class-specific mutators, such as mutators for the width and height of the Rectangle class in testRectangle.cpp, were tested as well.

Otherwise, a few other test files were created to test broadly applicable functionality. test3DRotation.cpp was created to test a Drawable object's ability to rotate via yaw, pitch, and roll about an arbitrary point, not just its own center coordinate. testDice.cpp was originally created to simulate rolling a random six-sided die, but ended up incomplete due to the revealed limitations of composite Euler rotations in 3D space.

Most test files were designed to be run multiple times. All of them included the required variables and test methods needed to properly ensure that all class functionality for each class was working as intended, but testing all functionality simultaneously tended to result in a visual mess, making an accurate assessment of whether things worked correctly more difficult to determine than necessary. It was much easier to determine whether an object was having its yaw altered appropriately when it was not simultaneously having its pitch, roll, and all three location coordinates mutated simultaneously while it changed color randomly every framebuffer refresh. Therefore, any mutators or accessors being called on the tested object were typically commented out in each test file, with perhaps one or two exceptions per file. The files were designed so that in order to test an object's functionality, one must first uncomment the appropriate methods, recompile the test file, and then run it as desired.

The final things changed were updated and added example files in accordance with the new library API. The prominent fix made was to testPhilosophers.cpp, which had its basic multithreaded graphical functionality restored for the new 3D Canvas. The biggest loss was the legend, which wasn't able to be reimplemented due to Image and Text still being unrestored, as they required texturing. Other added example files included testClock.cpp, which displayed a 3D time-accurate grandfather clock; testSolarSystem.cpp, which displayed a 3D model of

earth's solar system with planetary orbital periods that were relatively accurate to one another; test2Dvs3D.cpp, which displayed some of the advantages given by 3D functionality in the final showcase presentation; and testDiorama.cpp, which displayed the updated way that TSGL draws objects in the final showcase presentation.

Results and discussion

The final result of the project was an almost entirely functioning thread-safe graphical library with support for some level of both 2D and 3D object-oriented functionality. Two-dimensional capabilities allow the user to draw essentially any 2D polygon that they desire by calling the appropriate ConcavePolygon or ConvexPolygon constructor and passing in arrays of x and y vertices corresponding to the desired shape. However, if what they desire is not overly complex, TSGL also now offers the ability to place instances of more general 2D objects at any desired location and with any yaw, pitch, and roll within a 3D space. Fully reimplemented 2D classes include Arrow, Circle, ConvexPolygon, ConcavePolygon, Ellipse, Line, Polyline, Rectangle, RegularPolygon, Square, Star, Triangle, and TriangleStrip.

TSGL's three-dimensional capabilities are more limited, not allowing the user to draw arbitrary 3D shapes through arrays of vertex coordinates, but still exhibit a great deal of flexibility through the default classes that TSGL allows the user to utilize. Users can draw any of the provided 3D object classes to the Canvas, at any desired location and with any yaw, pitch, and roll within a 3D space. Fully implemented 3D classes include Cone, Cube, Cuboid, Cylinder, Ellipsoid, Prism, Pyramid, and Sphere.

All of these objects can be created extremely easily by the user with just a few parameters passed and added to the Canvas. Additionally, all of them provide access to instance variables that allow the user to freely mutate their location in the 3D space, their yaw, pitch, and roll, the point about which they rotate, and their color scheme. Most of these classes have instance variables which can be accessed and mutated in order to change their dimensions even after they've been added to the Canvas, provided the user has an appropriate pointer value which allows them to access the instance. The only classes which lack these dimensional mutators are classes which are originally highly customizable. There is no real catch-all way to modify the dimensions of an arbitrary ConcavePolygon or ConvexPolygon, or even a catch-all way to modify the dimensions of an arbitrary triangle. General mutator applicability is a tradeoff for increased user ability to specify their object.

The overall benefit that TSGL has gained has been one of ease of creation and mutation. As illustrated by test2Dvs3D.cpp and testDiorama.cpp in the final presentation showcase, drawing 3D objects to a Canvas is possible in 2D, with an extensive amount of time and calculation. Technically, after all the processing done by OpenGL on 3D objects in OpenGL, the final result is still just triangles, or even simpler, pixels on a two-dimensional screen. But the key difference is that in order to represent a 3D object with strictly 2D functionality, one has to draw a great deal of triangles individually, with either extensive trial and error or a great deal of calculation. But by being able to drop a 3D object into the 3D space and simply have OpenGL do all the necessary calculations, a great deal of time and effort is saved for the user in terms of creation. Additionally, if the user is unhappy with their creation, or desires to change it in any way over the course of the graphical animation, the energy spent on trial and error or

calculation must be respent in a 2D context. But in a 3D context, this energy can be saved again by achieving the same result in a few lines of code at most. This saves the library's user a great deal of time and effort in terms of mutation.

Conclusions

Overall, this project was very much successful. The perpetual tale with TSGL is gradual progress towards an improved end. It's not always perfect along the way, and no one who ever works on the library leaves it in a place where they're truly satisfied with it, but the important thing is that progress continues to be made, even if it's not always as much as everyone would like. After a student leaves, there is always another one to take up the mantle and continue improving the library. And at the end of the day, while cleaning up the installation scripts was an initial goal of this project that was never even touched, and TSGL lost the functionality to draw textured objects for now, there's no doubt that this project represents true progress for TSGL. An excellent amount of three-dimensional functionality was added, marking the transition into a new era in TSGL development. Sometimes it's necessary to take a few steps back in order to move forward, and despite having taken some steps back along the way, this project has undoubtedly moved forward overall.

Future Work

There are many things which could still be improved about TSGL. The library is not yet in what can be considered a stable implementation, and work will still have to be done in order to

bring it to a state where it can be merged into master and called stable. Here are seven improvements which would be extremely beneficial to the library's overall capabilities.

The first three improvements fall under the category of restorative work. First, reimplementing texturing capabilities would let TSGL be able to draw Text and Images once again, which are pretty important functionalities. The CS112 lab specifically uses TSGL's Text and Images to illustrate parallel processing capabilities, so restoring them is paramount before the 2020 fall semester. Adding texturing capabilities also might allow for some other interesting opportunities, like texturing the planets in the solar system example so that they're more graphically realistic.

Second, ensuring that all other Canvas functionality is restored from the changes made to it this year is equally important before the fall. Since Canvas::draw() has been completely replaced, things like screenshots and pixel sampling are likely currently not functioning as intended and will need to be updated. This is because both of these methods depended entirely on the pixel buffer being updated by Canvas::draw(), and the pixel buffer is not being currently updated by the newer, more spartan version of draw(). Screenshots and pixel sampling are also important for image processing, and therefore will be vital to restore before CS112 begins again.

Third, restoring some level of procedural drawing capabilities would be extremely useful. Even if they only served as essentially a painted background to be drawn before any other objects were drawn to the screen, this would allow for more interesting backgrounds than the bland single color backgrounds that TSGL currently is stuck with. Additionally, bringing

this functionality back would allow a great deal of test and example files which have been deprecated due to the changes in Canvas to be restored.

Fourth, once the previous changes have been made and TSGL has more or less had its old 2D capabilities restored in entirety, improving the rotation capabilities would be good. Whether that's using quaternions or rotating the object around an arbitrary axis rather than just the x, y, or z-axis with roll, pitch, or yaw respectively, in the current implementation TSGL's rotation capabilities are currently a bit lacking. This is due to the way composite Euler rotations work in 3D space. When one first applies roll, then pitch, then yaw, it results in a different rotation matrix than when one applies yaw, then pitch, then roll, since matrix multiplication is not commutative. At the bare minimum, there should be a way to alter an object's orientation at zero yaw, pitch, and roll, since each object currently only has one non-rotated orientation accessible. This can cause some serious problems if you'd like multiple objects to be oriented differently, but to rotate similarly. testDice.cpp illustrates the issue quite well.

The fifth improvement worth making is improving the transparency capabilities. Currently, TSGL just has a typical depth buffer, meaning that if an object is behind another transparent object, it won't be drawn, even if it should be visible through the transparent object. The "transparent cube" in testDiorama.cpp is in reality four squares with an empty top and bottom, in order that one can actually see the objects "inside." A basic Cube, even if it's completely transparent, will prevent the other objects from being rendered due to the depth buffer. This problem could be solved in a variety of ways. None will be perfect, but one idea would be to remove the depth buffer, draw opaque objects first, and then sort the transparent objects by their z-coordinates and add them back to front.

Sixth, allowing the user to control the camera placement and angle would allow for more dynamic animations. Currently, it's essentially locked in place, but camera actions like zooms and pans would all become possible if the Canvas' perspective could be altered by the user. Even quick cuts would become possible, resulting in a more potent graphical experience.

The final improvement to be made is cleaning up how TSGL installs onto various operating systems like Linux, MacOS, and Windows. This was one of the original goals for this project, but it never really even got off the ground. Chris Wieringa made a very nice Debian install package for Linux at the end of summer 2019, but the install scripts for MacOS and Windows still have massive issues in terms of dependencies and overall cleanliness. The Linux Debian package will also likely need to be updated and remade whenever the SeniorProjectIA branch is stable enough to be merged into master.

Acknowledgements

First, this project was principally made possible by all the work done in previous years by other Calvin University Computer Science students to make TSGL what it was before the project began. Thanks go out to Patrick Crain, Mark Vander Stel, Chris Dilley, Christiaan Hazlett, Elizabeth Koning, and Ian Adams. Second, a massive thanks is extended to Dr. Joel Adams for being this project's advisor and supervising all the summer research and senior projects involving this library over the years. This project, and this library, would not be what they are if not for him. Additionally, thanks to Dr. Keith VanderLinden for being the overall supervisor for all the senior projects in the 2019-2020 school year. His coordination skills were invaluable for keeping everything on track for all seniors involved. Finally, an extensive thanks to the Calvin

University Computer Science department as a whole for making this opportunity possible. Without the hours upon hours of patient work put in by this department over the last four years, this project would be merely a dream.

References

[1] “C OpenGL, GLFW Drawing a simple cube,” *Stack Overflow*, 04-Jun-2014. [Online]. Available: https://stackoverflow.com/questions/24040982/c-opengl-glfw-drawing-a-simple-cube. [Accessed: Mar-2020].

[2] “Concave Polygon via Stencil Buffer,” *Khronos Forums*, 10-Mar-2011. [Online]. Available: https://community.khronos.org/t/concave-polygon-via-stencil-buffer/63826/4. [Accessed: May-2020].

[3] “Creating a sphere,” *Khronos Forums*, 18-Mar-2012. [Online]. Available: https://community.khronos.org/t/creating-a-sphere/67128. [Accessed: Mar-2020].

[4] “glScale,” *OpenGL 2.1 Reference Pages*. [Online]. Available: https://www.khronos.org/registry/OpenGL-Refpages/gl2.1/xhtml/glScale.xml. [Accessed: Apr-2020].

[5] S. Guha, *Computer graphics through openGL®: from theory to experiments: comprehensive coverage of shaders and the programmable pipeline*, 3rd ed. Boca Raton: Taylor & Francis, a CRC title, part of the Taylor & Francis imprint, a member of the Taylor & Francis Group, the academic division of T&F Informa, plc, 2019.

[6] “Home of the Blender project - Free and Open 3D Creation Software,” *blender.org*. [Online]. Available: https://www.blender.org/. [Accessed: Oct-2019].

[7] S. M. LaValle, *Planning algorithms*. New York (NY): Cambridge University Press, 2014.

[8] “OpenGL - The Industry's Foundation for High Performance Graphics,” *The Khronos Group*, 19-Jul-2011. [Online]. Available: https://www.khronos.org/opengl/. [Accessed: Apr-2020].

[9] “Rotation Matrix of rotation around a point other than the origin,” *Mathematics Stack Exchange*, 11-Jan-2017. [Online]. Available:

https://math.stackexchange.com/questions/2093314/rotation-matrix-of-rotation-around-a-point-other-than-the-origin. [Accessed: Mar-2020].

Appendices

https://github.com/Calvin-CS/TSGL/tree/SeniorProjectIA