Tutorial 5: Texture Mapping

Note to the reader

> Your objective is to write code so that your application behaves similar to the sample or better. The tutorial is verbose only because it assumes minimal previous experience in computer graphics and the OpenGL toolbox. If you're so inclined, you can skip this tutorial and instead play with the sample, and then read the submission guidelines and rubrics to ensure your submission is graded fairly and objectively. Or, you can pick-and-choose which portions to read and which to ignore with the disadvantage that continuity and comprehension might be lost. Or, just read the entire tutorial and it is possible you may learn one or two things.

Topics Covered

- Understand texels, texel space, texture space, and texture coordinates
- Understand how to implement basic procedural texture mapping
- Understand how to implement ease-in/ease-out animation and its advantages over linear animation
- Understand how to implement basic texture mapping in OpenGL
- Understand how to use blending modes to implement transparency
- Understand wrapping/repeating, mirroring, and clamping OpenGL commands
- Understand how to combine colors from texture mapping and lighting
- Research specific OpenGL commands
- Research and explore GLSL to extend functionality of vertex and fragment shaders

Prerequisites

- You have read the lecture presentation on texture mapping
- You have completed the worksheets on transformations and texture mapping
- You have completed Tutorial 4

First Steps

Overwrite the existing batch file icg.bat in icg-opengl-dev with a new version available on the assessment's web page. Execute the batch file.

1. Choose option **5 - Create Tutorial 5** to create a Visual Studio 2022 project tutorial-5.vcxproj in directory build with source in directory projects/tutorial-5 whose layout is shown below:

```
icg-opengl-dev/      # OpenGL sandbox directory
├─ build/               # Build files will exist here
├─ images/              # Contain texture files(.tex)
├─ meshes/              # Mesh files (.msh, .obj) etc.
├─ scenes/              # Scene files (.scn)
├─ projects/            # Tutorials and assignments
|  └─ tutorial-5       # Tutorial 5 code exists here
|     └─ include       # Header files - *.hpp and *.h files
|     └─ src           # Source files - *.cpp and .c files
|     └─ shaders       # Shader files - .vert and .frag
|        └─ my-tutorial-5.vert # Vertex shader file
|        └─ my-tutorial-5.frag # Fragment shader file
└─ icg.bat           # Automation Script
```

Source and shader files are pulled into nested directory `./projects/tutorial-5/src` while header files are pulled into nested directory `./projects/tutorial-5/include`. This project make use of texture files for images that are pulled into directory `./images`.

2. Move the vertex and fragment shader source files from `/projects/tutorial-4/shaders/my-tutorial-4.*` to `/projects/tutorial-5/shaders/my-tutorial-5.*`.

Getting Started

1. The tasks in this tutorial require a single rectangular polygon to be rendered to the viewport. It does not require transforms or the concept of models, objects, and instancing from previous tutorials. You can start with source code from any previous tutorial.
2. Before starting the tutorial, execute the sample located at `sample/tutorial-5.exe`. Keyboard controls are provided in the window's title bar and are listed below:
    1. Keyboard $\mathbf{T}$ allows you to step through tasks 0 through 7.
    2. Keyboard $\mathbf{M}$ toggles modulate mode that combines per-fragment color and per-fragment texture color.
    3. Keyboard $\mathbf{A}$ toggles alpha blending mode.
    4. Keyboard $\mathbf{R}$ toggles rotation of texture coordinates.

Overview of Texture Mapping

As seen in Figure 1, shading objects with solid colors is nice enough but they're "plasticky" and quite bland. These uniform colors don't provide the rapid changes in colors across small areas that occurs in both naturally occurring surfaces such as wood and stone or man-made objects.

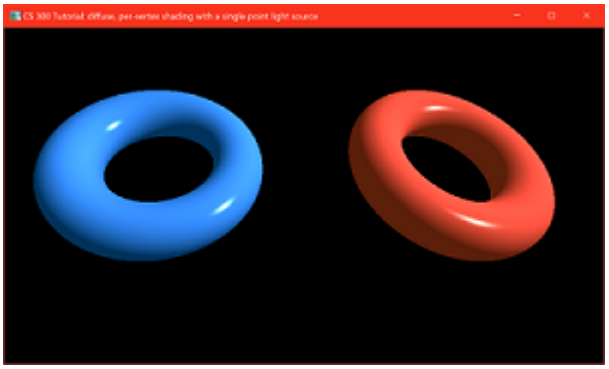

Figure 1: Objects rendered using Phong illumination model

Rendering high visual details representing irregular variations in color, roughness of surfaces, and imperfections caused by wear and tear requires data-intensive geometry with individual triangles modeling regions of linear color changes. Texture mapping is an incredibly powerful and inexpensive method for simulating complex geometry and adding details to surfaces by "gluing" an image onto a geometrically simple surface, such as an image of a brick wall on a flat rectangle. Texture mapping is so vital for the efficient creation of realistic images that modern graphics hardware are explicitly designed to efficiently implement the process. Figure 2 illustrates the idea of texture mapping.

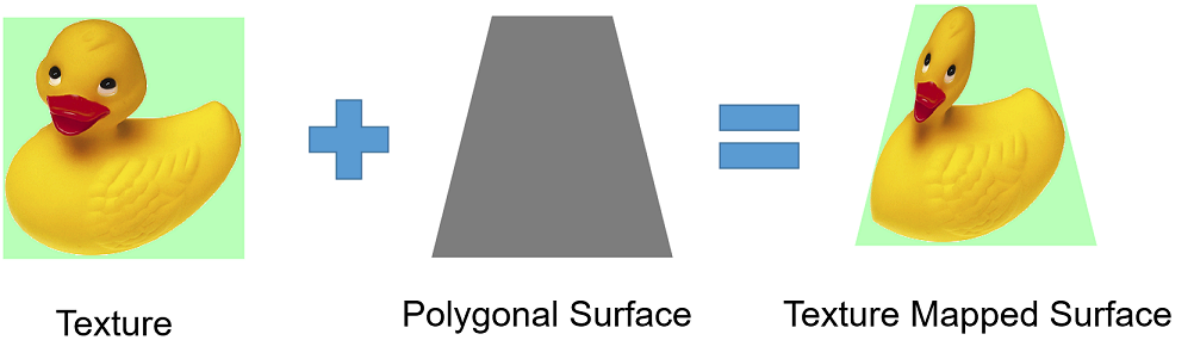

Figure 2: An overview of gluing images onto polygonal surfaces

Texture Images and Texels

In a process similar to gluing wallpaper to a surface or pasting a decal to a flat surface, texture mapping involves the mapping of an image called a ***texture*** to a simpler surface. Textures can be either one-, two-, or three-dimensional arrays of data. One-dimensional textures are suitable for linear primitives such as lines; two-dimensional textures are suitable for mapping images to planar surfaces; three-dimensional textures are volumetric allowing objects to appear as if carved out of solid material. This assignment is only concerned with two-dimensional textures. Every two-dimensional texture has a width w representing the number of image data samples in the horizontal direction and a height h representing the number of data samples in the vertical direction. Just as color buffers and screens consist of pixels, the atomic unit of each texture is called a texel [for **tex**ture map **el**ement]. Figure 3 illustrates the texel coordinate system for representing individual texels (x, y) of an image. Note that this $2-$tuple representation of a texel is the reverse of the notation for referencing the texels' corresponding memory location in C/C++. Figure 3 shows that - unlike many graphics formats and APIs - OpenGL identifies texels from bottom to top meaning that texels located at higher locations will have higher y values compared to texels located at lower locations.

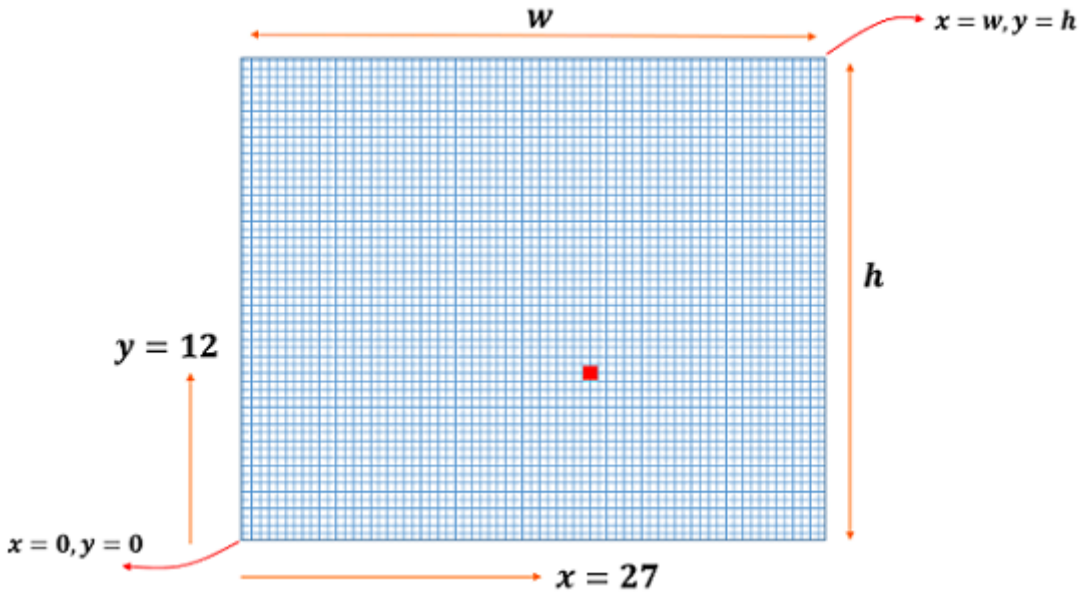

Figure 3: Texel coordinate space

In addition to the texture map's width and height, the other consideration is the format of data specified at each texel. This tutorial is only concerned with texels representing either color or color plus opacity. Color data is most often represented in **RGB color space**. We'll use $8-$bits for each channel for a total of $24-$bits to represent each texel. Color plus opacity data is represented in the so called **RGBA** color space with $\mathbf{A}$ representing opacity. In this case, we'll use $8-$bits for each channel for a total of $32-$bits to represent each texel. Somewhat confusingly, the opacity value is also referred to as **alpha**. Alpha value 0 [0.0 in floating-point] denotes full transparency while 255 [1.0 in floating-point] denotes full opaqueness with various degrees of translucency represented by alpha values in range $[0, 255]$ or $[0.0, 1.0]$ in floating-point. Certain texture mapping techniques will require textures to contain other types of data values including specular reflectance and diffuse reflectance coefficients, normal vectors to implement normal mapping or bump mapping, geometrical displacements of points on surfaces to implement displacement mapping, depth values to implement shadow mapping, or images computed at runtime to implement reflection mapping.

Texture Space and Texture Coordinates

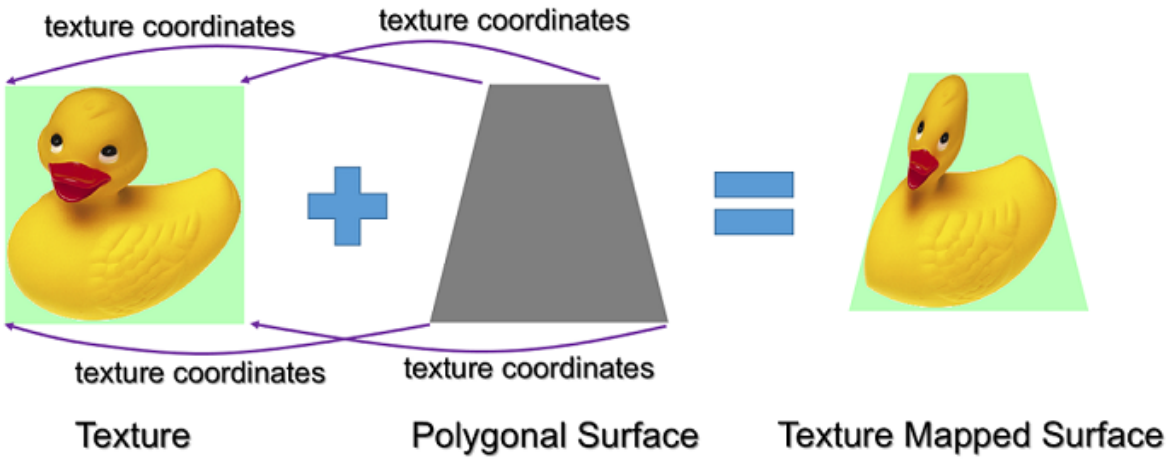

Figure 4: Understanding texture coordinates

When a texture is applied to a planar surface, each point on the surface has to correspond to a texel in the texture image. The earlier description of texture mapping as "gluing" a two-dimensional image onto a surface did not make clear the connection between surfaces and texture images and how texture images gets mapped onto surfaces. The correlation between the

texture image and how it appears as a texture when mapped onto the surface is provided by an extra vertex attribute called ***texture coordinates***. As shown in Figure 4, texture coordinates - specified at each vertex - control the texture's placement on the surface by determining which points [texels] in the texture image correspond to which points [vertices] on the surface.

Texture Space

Texture coordinates are described in a coordinate system called ***texture space***. Traditionally, the computer graphics community has referred to the horizontal and vertices axes of two-dimensional texture space as U and V, respectively. Thus, texture coordinates are also referred to as UV **coordinates** or UV for short. In contrast, OpenGL uses s to denote the horizontal axis ranging from left to right and t to denote the vertical axis from bottom to top. The unit square texture space $(s, t) \in [0, 1] \times [0, 1]$ is the mapping of discrete texel space to a continuous, normalized space that is independent of the two-dimensional texture image's width and height. Figure 5 illustrates the texture space associated with a texture image:

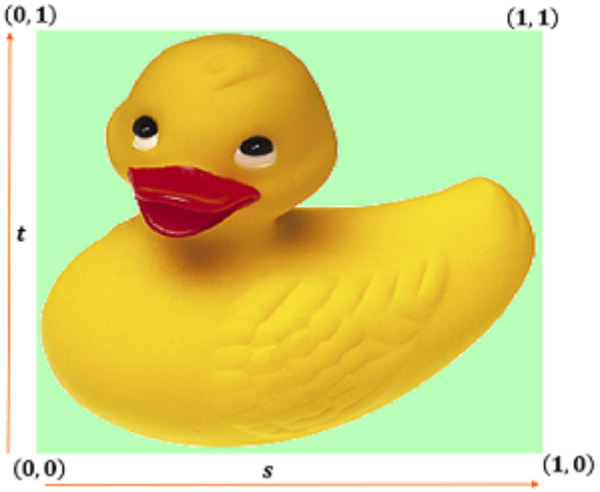

Figure 5: Texture coordinate space

Texture mapping in normalized texture space is more practical and efficient compared to texel space. Since texture coordinates are independent of the dimensions of the texture image, they don't need to be recomputed when texture images change. A value of 0.0 for s coordinate means "leftmost texel of the image", 0.5 means "middle texel", and 1.0 means "rightmost texel of the image". A value of 0.0 for t coordinate means "bottommost texel of the image" with 1.0 meaning "topmost texel." Values of s and t outside $[0, 1]$ are not inside the image, but such values are still valid as texture coordinates [more on this later].

Understanding Texture Coordinates

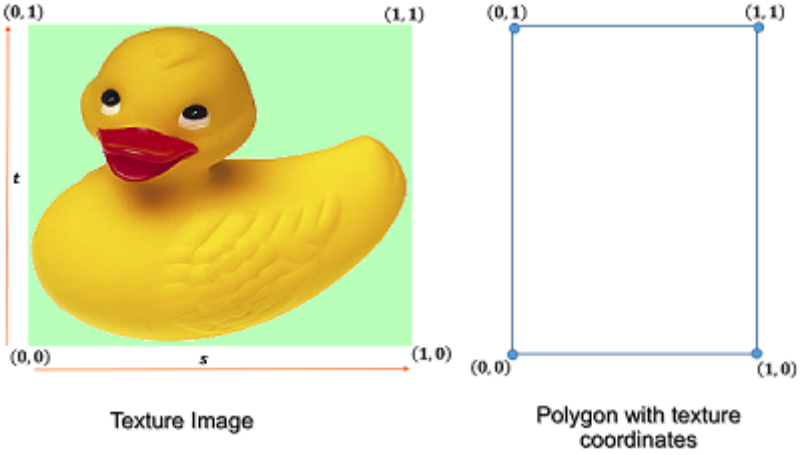

Figure 6: Associating texture coordinates to polygon vertices

To understand how all of this works, consider Figure 6. On the left, we've a texture image defined in texture space. On the right, we've a rectangular polygon onto which the texture image is to be pasted. Realize that in modern OpenGL this polygon is comprised of two triangles, but for this example, it doesn't matter. As described earlier, texture mapping requires every vertex to have an additional attribute called texture coordinates that describes which texel in the texture image the vertex maps to. The polygons are illustrated with texture coordinates and *not* position coordinates. It doesn't matter where the polygon is located in the $xy-$plane - there is no particular relationship between the position and texture coordinates of a vertex.

It doesn't matter that the texture image size doesn't match the "size" of the polygon - the image will be "stretched" or "squished" to fit the polygon. More correctly, though the image is not really stretched. Because the bottom-left polygon vertex has texture coordinates $(0, 0)$ it will map to the texture image's bottom-left texel. Since the top-right polygon vertex has texture coordinates $(1, 1)$, it will map to image's top-right texel.

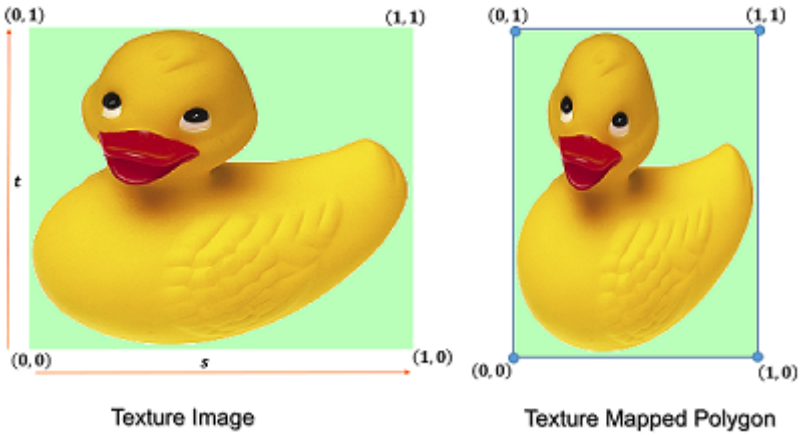

Figure 7: Interpolating texture coordinates across polygon surface

Figure 7 shows that the fixed-stage rasterizer will interpolate texture coordinates assigned to each vertex to compute the texture coordinates for interior points. These interpolated values will be between 0 and 1 depending on the relative position of the interior point. The fragment shader will receive the interpolated texture coordinates for a fragment location which are then mapped back into the texture image to compute the corresponding texel at that location.

🎯Task 0: Initial Setup [or, Painting Color on Surfaces]

1. Set up a vertex array object to render a rectangular polygon that exactly fits the viewport. You will need to extend the vertex array object [VAO] setup from previous tutorials to incorporate vertex position, color, and *texture* coordinates. We've been using *structure of arrays* format to define VAOs in previous tutorials. This tutorial will switch things and require vertex attributes to be buffered using ***array of structures*** format. Review $\text{Tutorial } 1$ if you've forgotten these two formats. The rectangle must be rendered with triangle primitives of type `GL_TRIANGLE_STRIP`.
2. Implement a vertex shader that takes position, color, and texture coordinate vertex attributes in generic vertex attribute slots 0, 1, and 2, respectively and passes them through to the next hardware stage unchanged.
3. Implement a fragment shader that takes interpolated color and texture coordinates as input. In this initial setup, the fragment shader will simply use the interpolated color from the rasterizer as the fragment's color.

4. The rendered image should resemble Figure 8. Your image may differ only in that you assigned different color values to the individual vertices. That is ok - the important thing is that you're able to render a rectangular polygon whose interior is shaded using interpolated vertex colors.

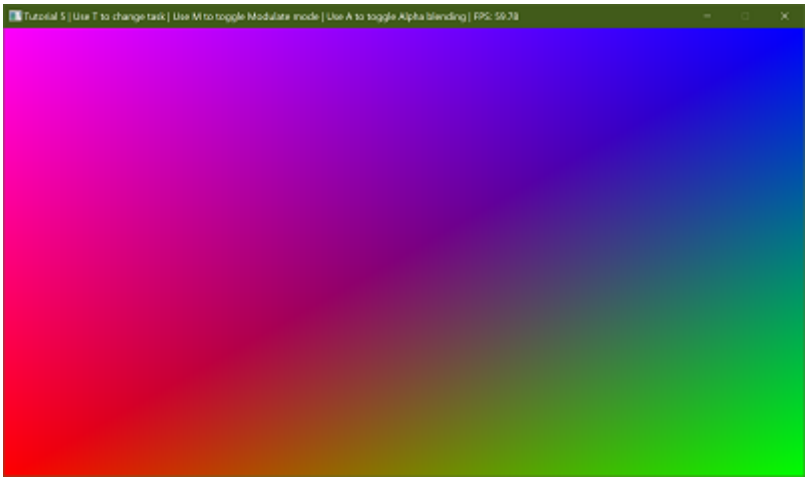

Figure 8: Polygon rendered with vertex color interpolation

5. Implement code that will allow the user to cycle through this and other tasks using keyboard button $\mathbf{T}$. One way to implement this feature is to associate an id with each task with the current task having id 0. The task id is sent to the shader program as an uniform variable and then used by the shader to execute code appropriate to the current task.

Procedural Texture Mapping

Procedural texturing refers to the use of code segments or algorithms to define color values in contrast to using color data from scanned-in images stored in files [aka texture maps]. One advantage of the procedural approach is lower GPU storage requirements since the complex details required to render a surface are abstracted into a procedure rather than explicitly specified in a two-dimensional array of texels. This is illustrated in Task 1. Since the details are no longer explicitly specified but implicit in the procedure, multiresolution textures can be created when required as shown in Task 2.

🎯Task 1: Procedurally generate fixed-sized checkerboard patterns

1. This task is concerned with generating a checkerboard pattern by adding code to the fragment shader. That is, if the two checkboard colors are $C_0 = (r_0, g_0, b_0)$ and $C_1 = (r_1, g_1, b_1)$, the fragment shader must determine whether a fragment with coordinates (x, y) is assigned color C_0 or color C_1. The simplest solution relies on the slices and stacks idea from $\mathrm{Tutorial\ 2}$. Suppose the viewport with dimensions $W \times H$ is sliced and stacked into $m \times n$ square tiles where each square tile has dimensions $size \times size$. Since the aim is to create a checkerboard pattern with alternating colors for adjacent tiles, an arbitrary tile (t_x, t_y) is assigned color C_0 if $(t_x + t_y)$ is even and assigned color C_1 if $(t_x + t_y)$ is odd:

$$c = (t_x + t_y) \bmod 2 \tag{1}$$

The tile (t_x, t_y) will have color C_0 if c evaluates to 0, otherwise it will have color C_1.

2. To evaluate Equation (1), we must know the viewport [or device] coordinates (x, y) of a fragment. The viewport coordinates $(x, y, z, 1/w)$ of a fragment can be obtained from a built-in variable named `gl_FragCoord`. Built-in variables are special variables that allow shaders to communicate with fixed stages of the pipeline and other shaders.

3. Given $size$, `gl_FragCoord.x`, and `gl_FragCoord.y`, it is left to the reader to modify the fragment shader to map the fragment's coordinates (x, y) to tile coordinates (t_x, t_y) and evaluate Equation (1). Hint: Given the width W of your viewport and tile size $size$, how many tiles does $size$ divide W into? To compute t_x, you only need to now determine which of the tiles does the fragment's x coordinate map to.

4. GLSL implements a large number of functions such as `dot`, `cross`, `cos`, `floor`, `ceil`, and `mod`. To obtain information about these and other GLSL function, use this page and click on the *use alternate (accordion-style) index* link located at the top-left corner of the page.

5. The sample uses $size = 32$, $C_0 = (1.0, 0.0, 1.0)$, and $C_1 = (0.0, 0.68, 0.94)$ to generate the checkboard pattern. The image generated by your code should match Figure 9. It is ok to have different values in your submission as long as the display is pleasing and creative.

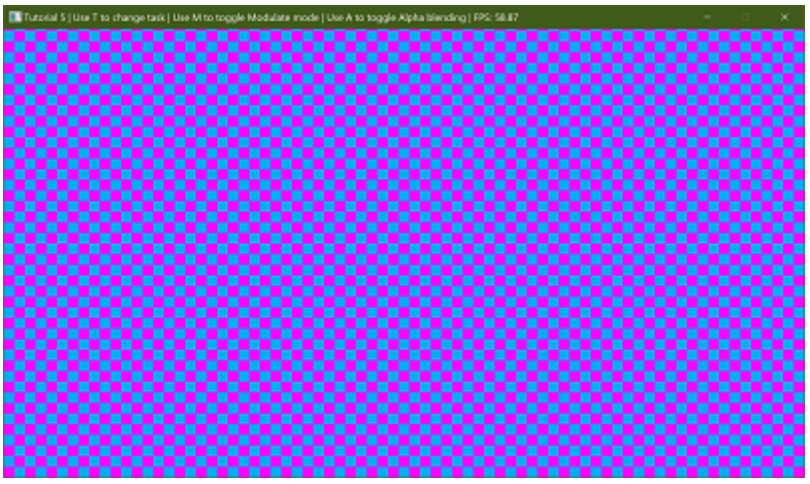

Figure 9: Generating fixed sized checkerboard pattern

6. The fragment shader is receiving interpolated vertex color attribute from the rasterizer while also procedurally generating a texture color at each fragment location. Set up a toggle controlled by keyboard button $\mathbf{M}$ to allow users to *modulate* the two colors with a component-wise product of the two colors. That is, modulating colors $C_0 = (r_0, g_0, b_0)$ and $C_1 = (r_1, g_1, b_1)$ gives a modulated color $C = (r_0 r_1, g_0 g_1, b_0 b_1)$. When programming GPUs, it is important to remember GLSL has vector types and can therefore handle vector operations such as scaling a vector, computing dot products and cross products as a single and atomic operation. Therefore, rather than multiplying corresponding components of the two colors, the modulated color should be computed as $C = C_0 * C_1$.

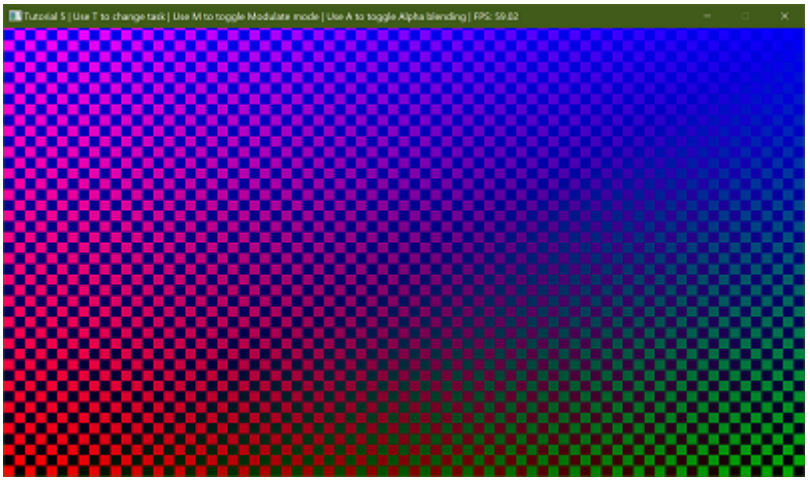

Figure 10: Modulating interpolated vertex color and procedural color

7. Since color components in the fragment shader are in range $[0.0, 1.0]$, the product of two such values will be smaller than or equivalent to both values. For example, think about the product of two positive numbers smaller than 1.0 such as 0.95 and 0.9; the resulting value 0.855 is smaller than both operands. The modulated image generated by your program should look similar to Figure 10.

🎯Task 2: Ease-in/ease-out animation of checkerboard size

1. The new task is to animate checkerboard tile size $size$ between values $[s_{min}, s_{max}]$. An easy and obvious way to implement the animation is to linearly increase $size$ from minimum value s_{min} by a constant amount Δs every time step or frame. After reaching maximum value s_{max}, $size$ is decreased by the same amount every time step. Suppose we wish to change the tile size from s_{min} to s_{max} through N frames. The linear interpolation equation to compute $size$ for current frame $f \in [0, N]$ would look like this:

$$t = \frac{f}{N}, f \in [0, N]$$
$$size = s_{min} + t\,(s_{max} - s_{min}), t \in [0, 1]$$

2. Time-dependent animation is more pleasing to the human eye than frame-dependent animation. Rather than using frame counts, suppose we wish to change the tile size from s_{min} to s_{max} through T seconds. The linear interpolation equation to compute $size$ for $time \in [0, T]$ would look like this:

$$t = \frac{time}{T}, time \in [0, T]$$
$$size = s_{min} + t\,(s_{max} - s_{min}), t \in [0, 1]$$

3. Function `update` in main.cpp calls helper function `GLHelper::update_time` to set the application's frame rate and the time in seconds spent by the application in executing the previous frame. This information is easily accessible and all of this means that it is fairly straightforward to implement time-dependent linear interpolation for this tutorial.

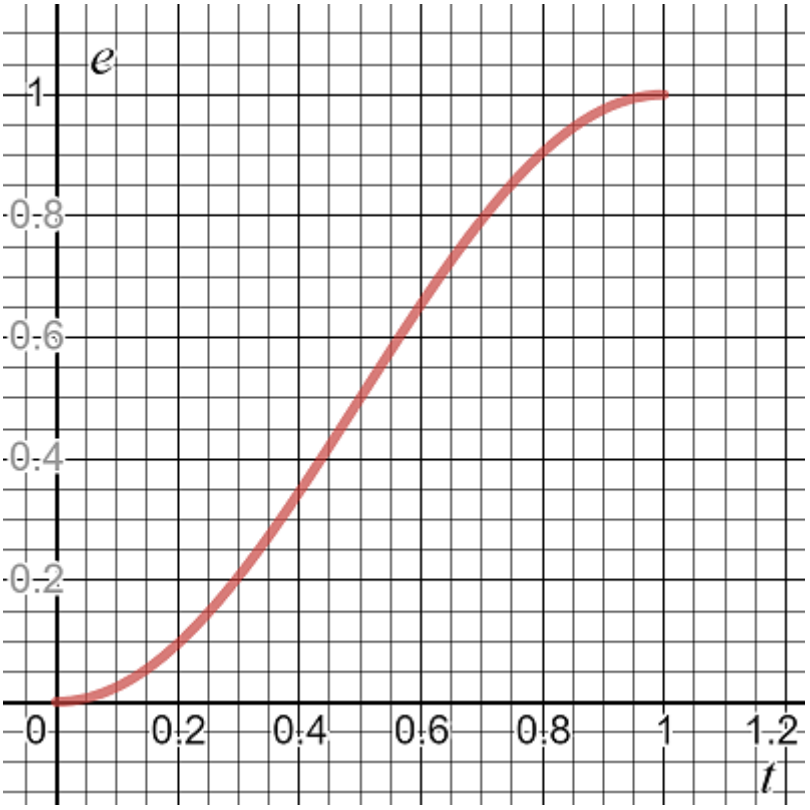

Figure 11: An example of function ease(t)

4. However, linear animations are unnatural, robotic, and jarring for game players. In contrast, easing-based animation provides smoother, more fluid, and realistic speed control. Ease-in animations start slow and end fast while ease-out animations start fast and end slow. An ease-in/ease-out animation provides a speed control that generates smooth changes to $size$ as it accelerates from start value s_{min}, reaches a maximum speed, and then decelerates to end value s_{max}. The process is reversed to animate $size$ from s_{max} to s_{min}. Ease-in/ease-out can be implemented by a function $e = ease(t)$ so that as $t = \frac{time}{T}$ varies uniformly between 0 and 1, e will start at 0, slowly increase in value and gain speed until the middle values, and then decelerate as t approaches 1. An example of function $ease(t)$ is illustrated in Figure 11 and the following equation:

$$\begin{aligned} t &= \frac{time}{T}, time \in [0, T] \\ e &= ease(t), t \in [0, 1] \\ size &= s_{min} + e\left(s_{max} - s_{min}\right), e \in [0, 1] \end{aligned}$$

5. One way to implement ease-in/ease-out is to use the $\sin$ curve section from $-\frac{\pi}{2}$ to $+\frac{\pi}{2}$ as the $ease(t)$ function. This is done by mapping parameter $t \in [0, 1]$ into value $t' \in [-\frac{\pi}{2}, +\frac{\pi}{2}]$ which is then used as the domain of the $\sin$ function. The range of the $\sin$ function is $[-1, 1]$ which is then mapped to range $e \in [0, 1]$ as shown in Figure 12 and Equation (2):

$$\begin{aligned} t &= \frac{time}{T}, time \in [0, T] \\ e &= \frac{\sin\left(\pi t - \frac{\pi}{2}\right) + 1}{2}, t \in [0, 1] \\ size &= s_{min} + e\left(s_{max} - s_{min}\right), e \in [0, 1] \end{aligned} \tag{2}$$

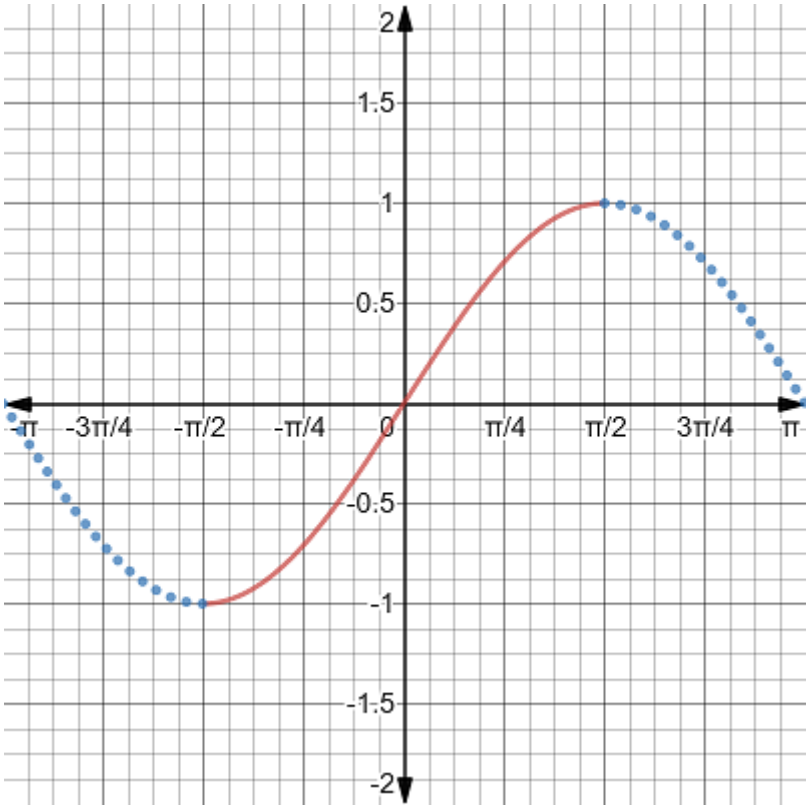

Figure 12: Sine curve segment (in red) to use as ease-in/ease-out control

6. Using the time spent in seconds by the application in the previous frame [available in `GLHelper::delta_time`], implement time-dependent [not frame-dependent!!!] ease-in/ease-out animation for $size$ on the CPU. The sample animates tile size $size$ from $s_{min} = 16$ to $s_{max} = 256$ in 30 seconds. Although the update time from one frame to the next is constant and uniform, the corresponding change in $size$ is non-linear and follows ease-in and ease-out principles. It is ok to have different values in your submission as long as the animation is pleasing and smooth.
7. Supply $size$ computed on the CPU in the previous step to the fragment shader as an uniform value, say `uTileSize`. The fragment shader will use the value in uniform variable `uTileSize` to map the fragment's coordinates (x, y) to tile coordinates (t_x, t_y) and compute fragment color C as in Equation (1). As before, the fragment's color is C_0 if e evaluates to 0, otherwise C_1. Run the sample to understand your submission's expected behavior.

Texture Mapping in OpenGL

There are various ways to go about applying a texture in OpenGL. In fact, textures were the most complicated part of OpenGL and have become more complicated in modern versions since they're so vital for the efficient creation of realistic images. This tutorial will constrain itself to implementing the most common variety where the texture image is two-dimensional and applied to a two-dimensional polygonal surface. In order to use texture mapping in OpenGL applications, the following steps must be implemented:

1. Create a texture object and load texel data into it.
2. Specify sampling parameters such as wrapping, filtering.
3. Include texture coordinates as an attribute to vertices.

4. Associate a texture sampler with each texture map used in a shader.
5. Retrieve texel values through the texture sampler in the fragment shader.

Loading image data from file and creating a texture object

A texture object is an OpenGL object that acts as a container for a set of images with the same texel format. Think of an image as a single array of texels of one-, or two-, or three-dimensions with a particular resolution, and a specific texel format. Just like other OpenGL objects that we've dealt with so far, texture objects simplify runtime interactions between the program and the OpenGL driver. Consider individually loading images from a set of images such as cubemaps and mipmaps. This is a cumbersome process for the programmer [because of the repetitive commands that must be coded] and an expensive operation that degrades runtime performance [because of the high cost of communication between client memory and GPU memory]. Once texture objects are individually initialized, they offer the possibility of using a single, fast OpenGL command to switch between sets of images.

The first step in initializing a texture object is to load image data from a file and use it to establish the texels for a texture object. Image data can be stored in many different file formats using varying compression schemes. Images can be of varying width and height. The image data loaded from file will have to saved to one of the many different internal formats used by OpenGL. Furthermore, OpenGL uses many different texture types [cube maps and mipmaps] to represent groups of images. To add to the complexity, OpenGL doesn't provide any functionality for loading images stored in commonly used image file formats. Instead, OpenGL programmers must write their own code to extract images from popular file formats such as as PNG, TIFF, JPG, TGA, and BMP. This is a non-trivial task because of the variety of compression techniques used in these file formats.

All of this means that the code for loading image data from files and storing the data in a texture buffer object can become quite complex depending on how generic the function should be. A popular option for OpenGL programmers is to rely on image loading libraries such as stb_image.h or FreeImage. Since the sole motivation of this exercise is to introduce texture mapping in OpenGL, an image loading library such as stb_image is not used to load images stored in file formats popular with artists. Instead, this exercise will use *raw* files that will not contain any header information - the file only contains texel data starting with the bottom row of texels and continuing up to the top-most row of texels in a manner consistent with OpenGL's convention for texel and texture space. If there is no header, how are the resolution and texel format of the image determined? We solve this problem by imposing the convention that our "raw" files will have image resolution 256×256 with each texel representing a $32-$bit RGBA value. Using this convention, we can now write a simplified image loading and texture object creation function called `setup_texobj` [in glapp.cpp]:

```
GLuint setup_texobj(std::string const& pathname) {
  // Directory images is the default repository for image files.
  // Hence, a call to this function would look like this:
  // GLuint texobj = setup_texobj("../images/duck-rgba-256.tex");

  // remember all our images have width and height of 256 texels and
  // use 32-bit RGBA texel format
  GLuint width {256}, height {256}, bytes_per_texel {4};

  // TODO: use the standard C++ library to open binary file, say
  // "../images/duck-rgba-256.tex" and copy its contents to heap
```

```
  // memory pointed by ptr_texels

  GLuint texobj_hdl;
  // define and initialize a handle to texture object that will
  // encapsulate two-dimensional textures
  glCreateTextures(GL_TEXTURE_2D, 1, &texobj_hdl);
  // allocate GPU storage for texture image data loaded from file
  glTextureStorage2D(texobj_hdl, 1, GL_RGBA8, width, height);
  // copy image data from client memory to GPU texture buffer memory
  glTextureSubImage2D(texobj_hdl, 0, 0, 0, width, height,
                      GL_RGBA, GL_UNSIGNED_BYTE, ptr_texels);
  // client memory not required since image is buffered in GPU memory
  delete[] ptr_texels;
  // nothing more to do - return handle to texture object
  return texobj_hdl;
}
/*
NOTE: Names are more descriptive than handles & std::map container will
allow programmers to conveniently reference specific texture objects by
name.
Although this exercise will use a single texture image stored in file
"../images/duck-rgba-256.tex", the function can be used in a general way to
initialize as many texture objects as required.
*/
```

Let's start to understand the most basic texture-related OpenGL commands that were used to create a texture object in the previous function. First, the following code fragment

```
GLuint texobj_hdl;
glCreateTextures(GL_TEXTURE_2D, 1, &texobj_hdl);
```

creates a texture object with name `texobj_hdl` with texture type `GL_TEXTURE_2D` representing a two-dimensional image and various texture parameters. Command `glCreateTextures` is a recent introduction to OpenGL since version 4.5: it is the equivalent of calling command `glGenTextures` followed by a call to command `glBindTexture`:

```
GLuint texobj_hdl;
glGenTexture(1, &texobj_hdl);
glBindTexture(GL_TEXTURE_2D, texobj_hdl);
```

So far, a texture object with name `texobj_hdl` was created to encapsulate two-dimensional textures and the image was loaded from file into heap memory with pointer `ptr_texels` pointing to the array of image data. The next step is to specify GPU storage and data for the two-dimensional texture encapsulated by the object. Each dimensionality of the texture object has an associated storage command: `glTextureStorage1D`, `glTextureStorage2D`, and `glTextureStorage3D` define the GPU storage for one-dimensional, two-dimensional, and three-dimensional textures, respectively. The following OpenGL 4.5 call provides GPU texture storage for texture object `tex_hdl` to store a two-dimensional texture image:

```
glTextureStorage2D(texobj_hdl, 1, GL_RGBA8, width, height);
```

Command `glTextureStorage2D` creates *immutable storage* for a two-dimensional texture image with storage size required to store all texels with *mipmap level* of 1 in `GL_RGBA8` *internal format* for each texel, and resolution given by `width` and `height`. The term *immutable storage* means that storage contents can be altered but not storage size and texel format. The OpenGL driver can make certain assumptions about this texture memory's usage and minimize overheads involved in dealing with the texture object's state. Mipmapping is a technique that uses a sequence of down-sampled images to reduce aliasing effects and increase rendering speeds when viewing texture mapped objects farther away from the camera. Since we're not mipmapping, there is only a *single* mipmap [the original texture map] and therefore the 2^{nd} argument is 1. The *internal format* of a texture is the format that OpenGL will use to internally store the texture data. If the image data is stored in client-side memory in a format different than specified to `glTextureStorage2D`, OpenGL will convert the data. OpenGL specifies a large number of internal formats and each comes with a size, performance, and quality tradeoff.

Once storage has been allocated for the image data in GPU's texture memory, the data must be loaded from client-memory into texture memory:

```
glTextureSubImage2D(texobj_hdl, 0, 0, 0, width, height,
                    GL_RGBA, GL_UNSIGNED_BYTE, ptr_texels);
```

`glTextureSubImage2D()` is an OpenGL 4.5 command that replaces a region of previously allocated GPU storage with new data. The remaining parameters provide details about the source image. The 2^{nd} argument specifies *mipmap level* - since we're not using mipmaps, the default level will be `0`. The 3^{rd} and 4^{th} arguments are texel offsets in horizontal and vertical directions specifying a location in texture memory from where to copy source texels. Since we're using the entire image, we use `0` for both arguments. The 5^{th} and 6^{th} arguments specify the source image's width and height. The 7^{th} and 8^{th} arguments specify the *format* determining composition of each texel in source image and *type* specifying the data type of each texel component in *format*. In our case, *format* is `GL_RGBA` and *type* is `GL_UNSIGNED_BYTE`. The final argument is the pointer to client memory where the source image is stored.

> ***OpenGL texturing tutorials tend to illustrate the creation of a two-dimensional texture image in GPU texture memory with calls to `glTexImage2D()`. This command requires many arguments, is easy to use incorrectly, difficult to use correctly, requires repeated calls to generate mipmaps, and sometimes leaves programmers with invalid textures. Instead, OpenGL 4.5 commands `glTextureStorage*()` and `glTextureSubImage*()` are recommended because they are designed to create texture images in GPU texture memory will less room for error.***

Texture parameters: Repeating textures

In earlier discussions, we've assigned each vertex an additional attribute called *texture coordinates*. The rasterizer interpolates these texture coordinates across the triangle's surface to determine texture coordinates for interior fragments. The fragment shader maps these interpolated texture coordinates to texel space to fetch the texels. This process is referred to as *sampling*. A multitude of parameters control different aspects of sampling. Rather than introduce each and every parameter, we examine a few parameters related to how sampling behaves when texture coordinates are *outside* the range $[0.0, 1.0]$. Why would texture coordinates be outside range $[0.0, 1.0]$? This commonly occurs when textures are applied to objects whose polygonal geometry is *larger* than texture map extents. There are three alternatives: *repeat* [or *tile* or *wrap*] the entire texture image, *mirror* the entire texture image, or *clamp* (or *extend* the texels at the edges of) the

texture image. Each of these options can be set separately along horizontal s and vertical t directions using OpenGL 4.5 command `glTextureParameteri`:

```
glTextureParameteri(tex_hdl, axis, mode);
```

where *axis* is one of `GL_TEXTURE_WRAP_S` or `GL_TEXTURE_WRAP_T` and *mode* is one of `GL_REPEAT` to *repeat* the entire texture map, `GL_MIRRORED_REPEAT` to *repeat* but *mirror* on every other repetition, or `GL_CLAMP_TO_EDGE` to *clamp* texture coordinate to 0 or to 1.

We've seen earlier that it doesn't matter that texture image size doesn't match polygon "size" - the image will be "stretched" or "squished" to fit the polygon. Now, consider Figure 13 below where we've a "small" image texture mapped onto a rectangular polygon four times "bigger" than the image. Notice that texture coordinates at the vertices are no longer in the range $[0.0, 1.0]$. Instead, the bottom-right, top-right, and top-left vertices have texture coordinates $(2, 0)$, $(2, 2)$, and $(0, 2)$, respectively.

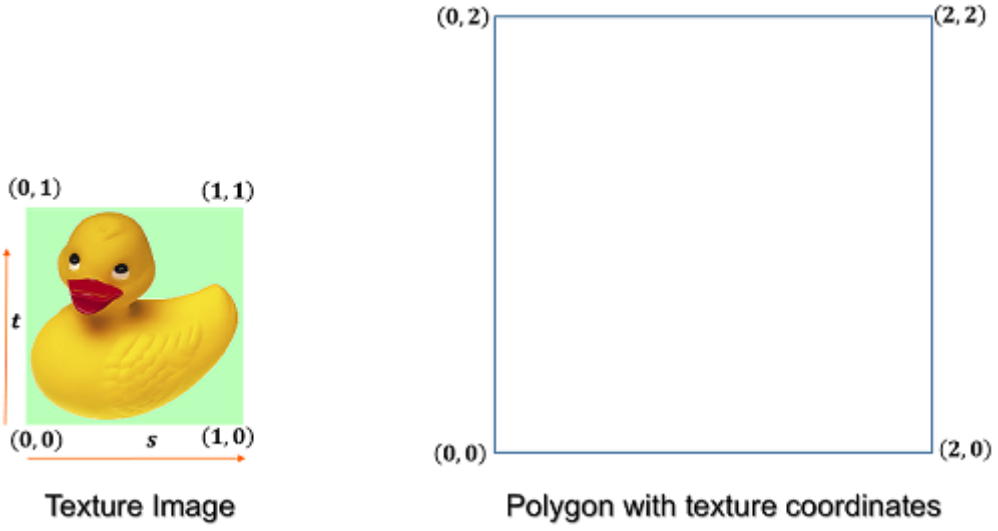

Figure 13: Setting texture coordinates for repeating images twice

Now, when we make these calls to `glTextureParameteri`:

```
glTextureParameteri(tex_hdl, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTextureParameteri(tex_hdl, GL_TEXTURE_WRAP_T, GL_REPEAT);
```

the "duck" is repeated twice across the polygon's horizontal and vertical directions. In particular, the texture image seems to repeat in every unit square of texture space with the integer portion of texture coordinates. It seems that as the polygon is mapped to a 2×2 rectangle in texture space with the texture image represented by a unit square in that texture space, it is painted with four copies of the texture image, as illustrated in Figure 14.

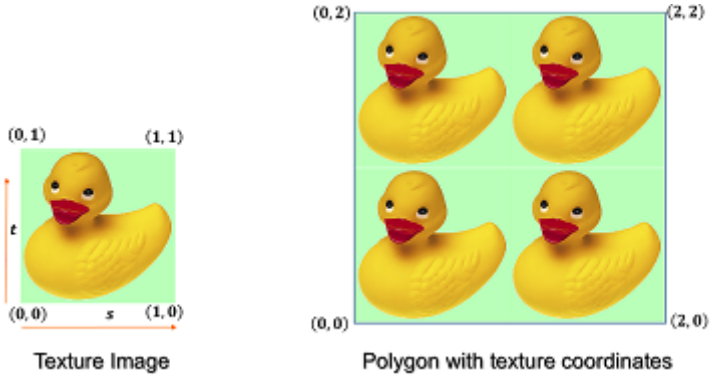

Figure 14: Repeating images twice along horizontal and vertical directions

This repeating of components s and t of a texture coordinate is mathematically defined as:

$$\begin{aligned} repeat(s) &= s - \lfloor s \rfloor \\ repeat(t) &= t - \lfloor t \rfloor \end{aligned} \tag{3}$$

Texture parameters: Mirroring textures

Let's now try to understand *mirroring*. When we make the following calls to `glTextureParameteri`:

```
glTextureParameteri(tex_hdl, GL_TEXTURE_WRAP_S, GL_MIRRORED_REPEAT);
glTextureParameteri(tex_hdl, GL_TEXTURE_WRAP_T, GL_MIRRORED_REPEAT);
```

for a rectangular polygon whose bottom-left, bottom-right, top-right, and top-left vertices have texture coordinates $(0, 0)$, $(4, 0)$, $(4, 4)$, and $(0, 4)$, respectively, the images are repeated across the polygon, but mirrored *every other repetition* in both horizontal and vertical directions. This behavior is shown in Figure 15.

Figure 15: Mirroring images along horizontal and vertical directions

The mirrored repeating of components s and t of a texture coordinate is mathematically defined as:

$$\begin{aligned} mirror(s) &= \begin{cases} s - \lfloor s \rfloor & : \lfloor s \rfloor \text{ is even} \\ 1.0 - (s - \lfloor s \rfloor) & : \lfloor s \rfloor \text{ is odd} \end{cases} \\ mirror(t) &= \begin{cases} t - \lfloor t \rfloor & : \lfloor t \rfloor \text{ is even} \\ 1.0 - (t - \lfloor t \rfloor) & : \lfloor t \rfloor \text{ is odd} \end{cases} \end{aligned} \tag{4}$$

Texture parameters: Clamping textures

The third option to deal with texture coordinates outside range $[0.0, 1.0]$ is to clamp the values to 0.0 or 1.0 - any texture coordinate value greater than 1.0 gets clamped to 1.0 and any texture coordinate value less than 0.0 gets clamped to 0.0. Clamping is set by making the following calls to `glTextureParameteri`:

```
glTextureParameteri(tex_hdl, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glTextureParameteri(tex_hdl, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
```

For a rectangular polygon whose bottom-left, bottom-right, top-right, and top-left vertices have texture coordinates $(0, 0)$, $(2, 0)$, $(2, 2)$, and $(0, 2)$, respectively, clamping has the effect of simply stretching the right and top edge or border texels across the entire polygon that lies outside the 1×1 unit texture coordinate square. This behavior is illustrated in Figure 16.

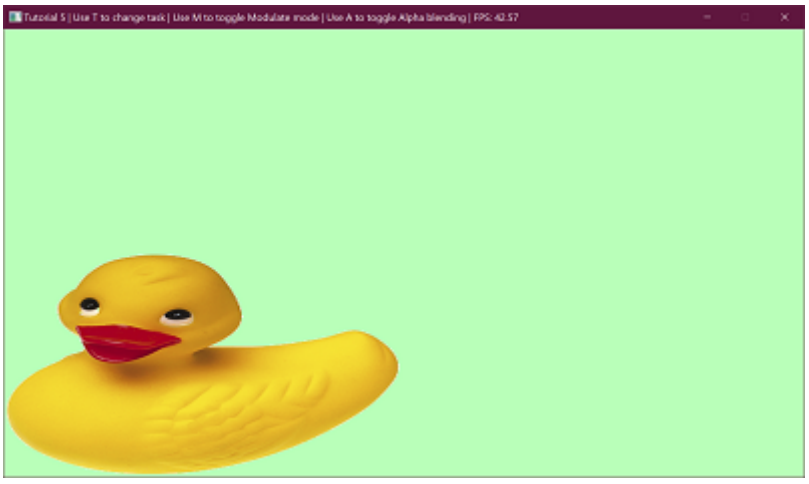

Figure 16: Clamping images along horizontal and vertical directions

The clamping of components s and t of a texture coordinate is mathematically defined as:

$$\begin{aligned} clamp(s) &= \max(\min(s, 1.0), 0.0) \\ clamp(t) &= \max(\min(t, 1.0), 0.0) \end{aligned} \tag{5}$$

Accessing texture maps from shaders: Texture image units and samplers

There are two remaining steps required to render a texture mapped rectangle: set up the application's texture state correctly before executing a shader that samples a texture image and set up a fragment shader to sample the texture image in a texture object. Although our discussion samples a texture image from a fragment shader, it doesn't mean vertex shaders can't sample textures!!!

We first begin by writing an example fragment shader for a simple texture mapping example:

```
#version 450 core
// incomplete fragment shader ...
layout(location=1) in vec2  vTexCoord;
layout(location=0) out vec4 fFragClr;
uniform sampler2D           uTex2d;

void main() {
  fFragClr = texture(uTex2d, vTexCoord);
}
```

A shader must use a `uniform` variable of type `sampler` to access a texture image. Think of a sampler object as an opaque data structure that is set up by the application so that it can access the appropriate texture image. The sampler type indicates the type of texture image to be accessed. A variable of type `sampler1D` accesses a one-dimensional texture image; a variable of type `sampler2D` accesses a two-dimensional texture image; a variable of type `sampler3D` accesses a three-dimensional texture image; a variable of type `samplerShadow2D` accesses a two-dimensional depth texture image; a variable of type `samplerCube` accesses a cube map texture image, and so on.

The fixed-stage rasterizer writes interpolated s and t texture coordinates values for a fragment in user-defined `in` variable `vTexCoord`. These texture coordinates are used to look up a value from a texture image bound to sampler variable `uTex2d` using GLSL built-in function `texture`. The first argument to `texture` is a sampler while the second argument is the fragment's texture coordinates (s, t). The type of the texture coordinate must match the sampler type - that is, if the sampler is of type `sampler2D`, the corresponding texture coordinate in the `texture` call must have type `vec2`. The hardware core implementing the fragment shader uses these texture coordinates to determine locations to be accessed in the texture image. The resulting texel is then used to create the final fragment color.

Suppose the rectangular model being rendered is encapsulating the texture object handle returned by a call to function `setup_texobj("../images/duck-rgba-256.tex")` in a variable `texobj`. How is texture object `texobj` attached to sampler variable `uTex2d`? This is done in the application and we'll look into that in the following code fragment extracted from a `draw` function that renders a texture mapped rectangle:

```
// suppose texture object is to use texture image unit 6
glBindTextureUnit(6, texobj);

// set what happens when sampler accesses textures outside [0, 1]
glTextureParameteri(texobj, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTextureParameteri(texobj, GL_TEXTURE_WRAP_T, GL_REPEAT);

// suppose shdrpgm is the handle to shader program object
// that will render the rectangular model
glUseProgram(shdrpgm);

// tell fragment shader sampler uTex2d will use texture image unit 6
GLuint tex_loc = glGetUniformLocation(shdrpgm, "uTex2d");
glUniform1i(tex_loc, 6);

// TODO: pass other uniform variables (if required) to fragment shader ...

// TODO: draw rectangular polygon ...

glUseProgram(0);
```

The next few paragraphs are a bit like a history lesson and are based on my experiences using OpenGL since about 1994. Although short, this background information will give you a good understanding of how samplers came about and the purpose they serve. Let's begin by looking at the evolution of OpenGL's texture mapping capabilities starting with OpenGL 1.0 which was released in 1994. Only one texture was supported by graphics hardware and the way to change it was to reload a new texture map from client memory to graphics hardware. This is completely wasteful because the previous texture must be reloaded again from client memory when it is reused for rendering. Even though there could be only one texture used at a time, there was still the need to create data for a one-dimensional texture, two-dimensional texture, and so on. To distinguish between different types of textures, OpenGL came up with the idea of *texture target*. Programmers could use calls to command `glEnable` to determine which texture target would be used in a rendering operation as seen in this code fragment:

```
// Defunct OpenGL code circa 1998 ...

```

```
// set up matrix stack ...
glLoadIdentity();
glTranslatef(20.f, 30.f, 40.f);

// use a two-dimensional texture map pointed to by ptr_tex_data
glEnable(GL_TEXTURE_2D)
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, 256, 256, 0, GL_RGBA,
             GL_UNSIGNED_BYTE, ptr_tex_data);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);

// render textured triangle
glBegin(GL_TRIANGLES)
    glTexCoord2f(0.0f, 0.0f)
    glVertex3f(-1.0f, -1.0f,  1.0f);
    glTexCoord2f(1.0f, 0.0f);
    glVertex3f( 1.0f, -1.0f,  1.0f);
    glTexCoord2f(1.0f, 1.0f);
    glVertex3f( 1.0f,  1.0f,  1.0f);
glEnd()
```

Most texture mapping applications switch among many different textures during the course of rendering a scene. To facilitate efficient switching among multiple textures and to facilitate efficient texture management, OpenGL 1.1 - released in 1997 - introduced texture objects to maintain texture state in GPU texture memory. The state of a texture object contains all of the texture images in the texture [including all mipmap levels] and the values of the texturing parameters that control how texels are accessed and filtered. Texture objects are distinguished by names generated by call(s) to `glGenTextures`. The application can have as many texture objects as required as long as there is enough memory available in the graphics hardware. Calling `glBindTexture` binds the named texture object, making it the current texture for the specified texture target. The target corresponds to texture type such as `GL_TEXTURE_1D`, `GL_TEXTURE_2D`, `GL_TEXTURE_CUBE_MAP`, and so on. When texturing is enabled, the current texture object is used for rendering. When rendering geometric objects using more than one texture, `glBindTexture` can be used to switch between these multiple textures. Switching textures is a fairly expensive operation. Even if the texture is already loaded, caches that maximize texture performance many be invalidated when switching textures. The details of switching a texture vary with different OpenGL implementations, but it is safe to assume that an OpenGL implementation is optimized to maximize texturing performance for the currently bound texture, and that switching textures must be minimized.

OpenGL 1.3 extended core texture mapping capability by providing a framework to support two or more textures to be applied to a fragment in one texturing pass. This feature came about because graphics hardware began to incorporate multiple texture image units. A *texture image unit* is silicon in GPUs that performs the sampling and interpolation of texel data in a texture. It has the ability to independently access, filter, and supply its own texture color to each rasterized fragment. OpenGL's multitexture support requires that every texture image unit be fully functional and maintain state that is independent of all other texture image units. The number of texture image units available in a given OpenGL implementation can be found by querying the implementation-dependent constant `GL_MAX_COMBINED_TEXTURE_IMAGE_UNITS` using `glGetIntegerv`. The OpenGL multitexture API added commands that control the state of one or more texture image

units. Command `glActiveTexture` determines which texture image unit will be affected by the OpenGL texture commands that follow. Since the active or current texture image unit can handle textures of many types [such as `GL_TEXTURE_1D`, `GL_TEXTURE_2D`, and so on], the texture target to be used in subsequent rendering operations is enabled using `glEnable`.

A texture image unit must access a texture object to do its work. Think of a texture image unit as the processor while the texture object holds the data to be processed. Therefore, the texture object that is to be used for texel lookup must be bound to the texture image unit with a call to `glBindTexture`. Backward-compatibility was achieved by making texture image unit 0 be the default texture image unit. This is what happens in our texture initialization function `setup_texobj` - even though `glActiveTexture` is not called, texture object `texobj_hdl` is bound to texture image unit `0`. In the following OpenGL 1.3 code fragment, the commands to enable two-dimensional texturing on texture image unit `0` and one-dimensional texturing on texture image unit `1` are shown:

```
// Defunct OpenGL code circa 2001 ...

// texture object names defined in global scope ...
GLuint texobj_hdl0, texobj_hdl1;

// initialize texture objects in init() ...

// this is extract from draw() ...

// set up texture image unit 0 to sample 2D texture image in tex_hdl0
glActiveTexture(GL_TEXTURE0); // set texture image unit 0 as current
// make texture image unit 0 use 2D texture image in tex_hdl0
glBindTexture(GL_TEXTURE_2D, texobj_hdl0);
// enable texture image unit 0 for sampling 2D texture image
glEnable(GL_TEXTURE_2D);

// set up texture unit 1 to sample 1D texture image in tex_hdl1
glActiveTexture(GL_TEXTURE1); // set texture image unit 1 as current
// make texture image unit 1 use 2D texture image in tex_hdl1
glBindTexture(GL_TEXTURE_1d, texobj_hdl1);
// enable texture image unit 1 for sampling 1D texture image
glEnable(GL_TEXTURE_1D);

// specify two sets of texture coordinates for each vertex
glBegin(GL_TRIANGLES);
  glMultiTexCoord2f(GL_TEXTURE0, 0.f);
  glMultiTexCoord2f(GL_TEXTURE1, 1.f, 0.f);
  glVertex3f(...);
  ... // you kinda get the idea
glEnd();
```

Let's fast forward to OpenGL 2.0 that introduced GLSL, vertex shaders, and fragment shaders. Texture lookups require the selection of a texture image unit and a texture image for the texture unit to lookup. First, `glActiveTexture` is called to make a texture image unit current so that subsequent texture calls will affect this texture image unit. Next, the texture image unit is enabled for a certain texture type with command `glEnable`. Finally, command `glBindTexture` provides the texture image unit access to a texture object. Managing the underlying implementation of texture image units and other texture lookup hardware is OpenGL's responsibility - GLSL doesn't

care about this stuff. Hence, GLSL provides a special opaque handle to encapsulate what to look up. These handles are called *samplers*. GLSL provides a variety of sampler types each of which can look up a particular texture type. A variable of type `sampler1D` can be used to access a one-dimensional texture map with target `GL_TEXTURE_1D`, `sampler2D` can be used to access a two-dimensional texture map with target `GL_TEXTURE_2D`, and so on.

Shaders cannot themselves initialize samplers. They can only receive them from the application, through a `uniform` qualified sampler, or pass them to user-defined or built-in functions. The sampler's location needs to be queried with `glGetUniformLocation`, just like any uniform variable. Sampler values need to be set by calling `glUniform1i`. What value should the application use as an initializer for a sampler? Samplers must be able to identify the texture object used for texture lookup. It may seem obvious that the application would initialize the sampler with the texture object representing the texture image to lookup. Instead, OpenGL's design requires the application to initialize the sampler with a value indicating the texture image unit being accessed. It is the application's responsibility to ensure this texture image unit is active and the texture object to be looked up is bound to this texture image unit. Since the samplers' type will match the texture units' texture target used in the texture unit [e.g., shader variable of type `shader2D` can only look up two-dimensional texture images attached to target `GL_TEXTURE_2D`], OpenGL has deprecated the requirement to call the command `glEnable`. To complete the setup, OpenGL 4.5 provides the convenient command `glBindTextureUnit` to bind the rectangular model's texture object `texobj` with texture unit numbered `6`:

```
1  glBindTextureUnit(6, texobj);
```

Let's recap our knowledge of texture objects, image units, targets, and sampler by looking at fragment shader code and extract from a draw call. First, here's the fragment shader code:

```
1  #version 450 core
2  // incomplete fragment shader ...
3  layout(location=1) in vec2  vTexCoord;
4  layout(location=0) out vec4 fFragClr;
5  uniform sampler2D           uTex2d;
6
7  void main() {
8    fFragClr = texture(uTex2d, vTexCoord);
9  }
```

The fragment shader code should now be straightforward to understand. Line 3 indicates that the shader is receiving interpolated texture coordinates from the rasterizer. Line 4 indicates that we're writing out the fragment's color in the fragment shaders' color buffer 0. Line 5 defines a uniform variable for *sampling* two-dimensional texture images with *target* `GL_TEXTURE_2D`. It is the application's job to ensure that the integer value assigned to `uTex2d` is the texture image unit that is *active* and has a texture object bound to the texture image unit's `GL_TEXTURE_2D` target. Line 8 calls built-in function `texture` to look up texels in a two-dimensional texture image encapsulated by the texture object bound to the *active* texture image unit [provided by the first argument] using the fragment's texture coordinates `vTexCoord` as the second argument. The color value returned from looking up the texels is sent down the hardware pipe to be further processed by the *per fragment processing* stage.

Now, lets review the setup from the application's side to ensure that texture lookups by the fragment shaders' sampler are successful.

```
1  // texture object texobj will use texture image unit 6
2  glBindTextureUnit(6, texobj);
3
4  // set what happens when sampler accesses textures outside [0, 1]
5  glTextureParameteri(texobj, GL_TEXTURE_WRAP_S, GL_REPEAT);
6  glTextureParameteri(texobj, GL_TEXTURE_WRAP_T, GL_REPEAT);
7
8  glUseProgram(shdrpgm); // enable shader program
9
10 // tell fragment shader sampler2D uTex2d will use texture image unit 6
11 GLuint tex_loc = glGetUniformLocation(shdrpgm, "uTex2d");
12 glUniform1i(tex_loc, 6);
13
14 // TODO: pass other uniform variables (if required) to fragment shader ...
15
16 // TODO: draw rectangular polygon ...
17
18 glUseProgram(0);
```

Line 2 makes a call to `glBindTextureUnit` to bind the texture object `texobj` [which is assigned the handle returned by call `setup_texobj("../images/duck-rgba-256.tex")`] to texture image unit `6`. Lines 5 and 6 specify parameters to the texture object that we want the texture image to be *repeated* if the texture coordinates are outside the unit texture space. Line 8 enables the shader program. Lines 11 and 12 initialize the uniform sampler `uTex2d` of type `sampler2D` with the texture image unit number `6` to which the texture object `texobj` is bound. Inside the fragment shader, `uTex2d` will lookup the texture image encapsulated by texture object `texobj`.

🎯Task 3: Rendering a texture map on rectangular polygon

Based on our discussion so far, the following is assumed:

1. The expression `setup_texobj("../images/duck-rgba-256.tex")` has been evaluated to return a texture object that is assigned to a texture object handle labeled `texobj`. This texture object encapsulates image data with width and height of 256 texels with each texel having $32-$bit *internal format* `GL_RGBA`.
2. A rectangular polygon is defined with position, color, and texture coordinates attributes at each vertex with a VAO defined using array of structures format. This rectangular polygon is to be rendered using `GL_TRIANGLE_STRIP` primitives. Texture coordinates at bottom-left, bottom-right, top-right, and top-left vertices have the following values: $(0, 0)$, $(1, 0)$, $(1, 1)$, and $(0, 1)$, in that order. Let's assume `vaoid` is the name of the handle to the rectangular object's VAO.
3. The shader program handle from Task 2 has name `shdrpgm`.
4. An understanding of wrapping behaviors when sampling a texture and how to set up these wrapping behaviors by calls to OpenGL command `glTextureParameteri`.
5. An understanding of how to set up the application's texture state correctly before executing a shader that samples a texture image.
6. An understanding of the fragment shader discussed in the previous section that samples a texture image.

Your task is to apply the "duck" texture map encoded in raw image file ../images/duck-rgba-256.tex on the rectangular polygon. Your application should generate an image shown in Figure 17.

Figure 17: Decal mode - gluing texture image on to polygon

The user can toggle the modulate mode by pressing the keyboard button M. The image generated by your program when modulating colors should match Figure 18.

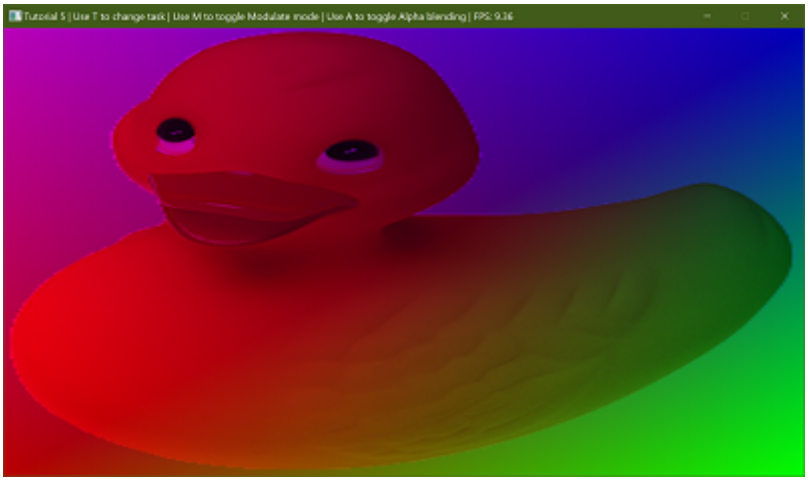

Figure 18: Modulating texture colors and interpolated vertex colors

Alpha Blending

Once the fragment shader computes a fragment's color, the fragment [with the newly added color] is sent to the next stage in the hardware pipeline called the *Per Fragment Operations*:

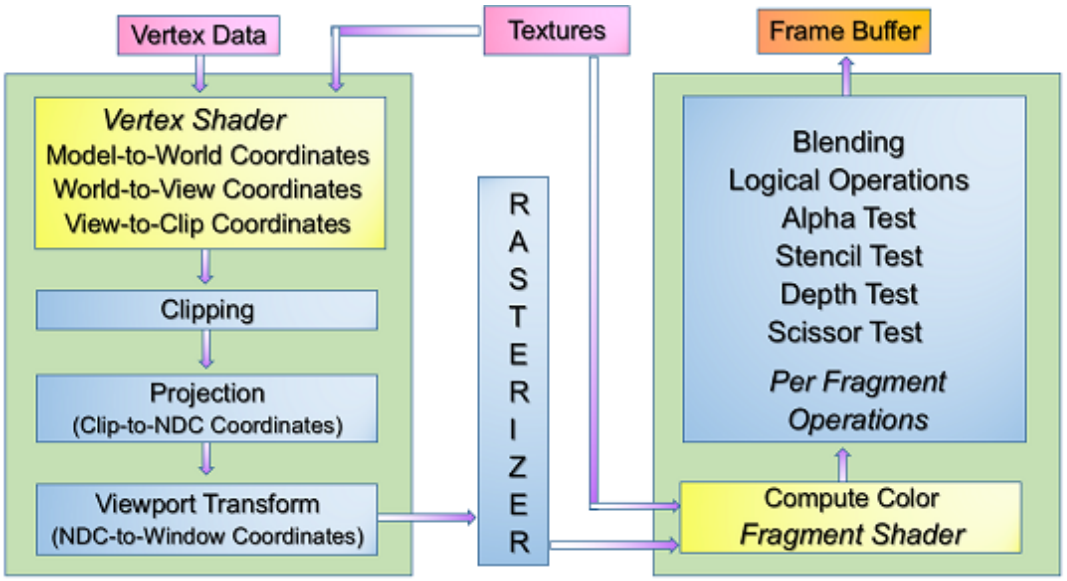

Figure 19: Simplified graphics pipe illustrating per fragment operations

If the fragment passes all enabled tests, it can be combined with the corresponding color value in the color buffer [the term framebuffer is more general in the OpenGL context - it includes the double-buffered color buffers and depth buffer]. This process is called blending. One way to implement basic transparency and translucency effects is to blend the fragment color and the color buffer color using the fragment's alpha value as the blend factor. Alpha value is a measure of translucency and it is commonly used to simulate translucent surfaces. We say that a fragment is *transparent* if its alpha value is 0; *opaque* if the alpha value is 1; and various degrees of *translucency* for all other in-between values. Recall that the texel format in the "duck" texture image is $32-$bit RGBA with the fourth component referred to as the *alpha value*. I hacked the image such that the "greenish" parts of the "duck" image have an alpha value of 0 while the rest of the image has an alpha value of 1. Alpha values read by the fragment shader sampler from the texture image can be used to render the rectangular polygon to be transparent wherever the "duck" image is greenish and opaque everywhere else.

To enable blending in OpenGL, you have to make a call to `glEnable` with `GL_BLEND` as the argument. Next, you determine how to blend source fragment color S with the destination pixel color D [from the color buffer at the fragment's location] using command `glBlendFunc`. Equation (6) illustrates the combination of S and D colors to implement transparency effects with the source fragment alpha A_S behaving as the interpolant. When A_S is 1 [that is, the surface being rendered is opaque], then we can't see through the opaque surface and therefore the fragment color is written to the color buffer. On the other hand, when A_S is 0 [that is, the surface being rendered is transparent], then the destination color in the color buffer will remain unchanged. For all other $A_S \in (0, 1)$, evaluation of Equation (6) will determine the color buffer color as a blend of the source and destination colors:

$$A_S \begin{bmatrix} R_S \\ G_S \\ B_S \\ A_S \end{bmatrix} + (1 - A_S) \begin{bmatrix} R_D \\ G_D \\ B_D \\ A_D \end{bmatrix} = \begin{bmatrix} A_S R_S + (1 - A_S) R_D \\ A_S G_S + (1 - A_S) G_D \\ A_S B_S + (1 - A_S) B_D \\ A_S A_S + (1 - A_S) A_D \end{bmatrix} \tag{6}$$

The user can toggle the transparency mode by pressing the keyboard button $\mathbf{A}$. The image generated by your program with the transparency mode should match Figure 20.

Figure 20: Blending between texture color and color buffer color

Note that alpha blending and color modulation are independent of each other. Execute the sample to view the correct image that your program should generate when keyboard buttons $\mathbf{M}$ and $\mathbf{A}$ are separately toggled.

🎯Task 4: Repeating textures

For this task, the application must do the following:

1. In the ***vertex shader*** [why not the fragment shader?], scale texture coordinates by a value of 4 before they are passed to the rasterizer. Try other values such as 2, or 8, and so on.
2. Set texture parameters to `GL_REPEAT` mode along both horizontal and vertical directions.

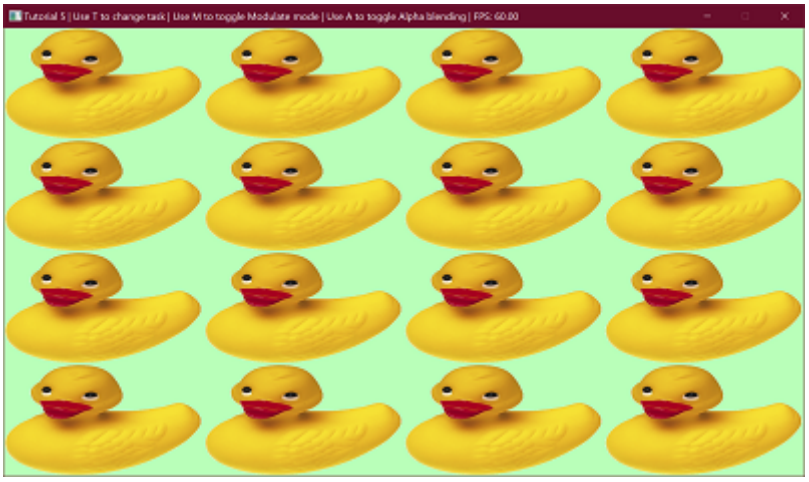

Figure 21: Repeating textures

The image generated by your program should match Figure 21.

The user can toggle color modulate mode and alpha blending by pressing the keyboard button $\mathbf{M}$ and keyboard button $\mathbf{A}$, respectively. Figure 22 shows the image that should be generated by your program when modulating texture and interpolated vertex colors.

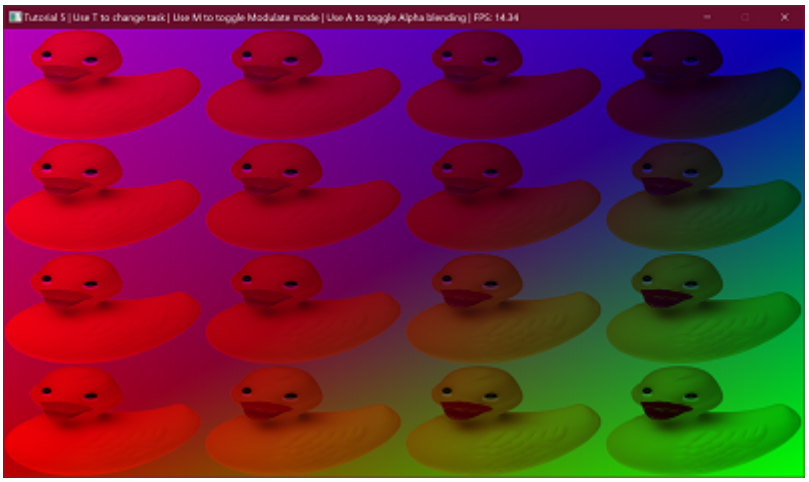

Figure 22: Repeating textures with modulated fragment color

Execute the sample to view the image that should be generated by your program when separately toggling keyboard buttons $\mathbf{M}$ and $\mathbf{A}$.

🎯Task 5: Repeating and mirroring textures

For this task, the application must do the following:

1. In the ***vertex shader*** [again why not the fragment shader?], scale texture coordinates by a value of 4 before they are passed to the rasterizer. Again, experiment with other values.
2. Set the texture parameters to `GL_MIRRORED_REPEAT` mode along both horizontal and vertical directions.

Figure 23 illustrates repeated and mirrored texturing of an image across the surface of the rectangular polygon - your picture must match the image in Figure 23.

Figure 23: Repeating and mirroring textures

The user can toggle the modulate mode by pressing the keyboard button $\mathbf{M}$. The image generated by your program when modulating colors should match the image in Figure 24.

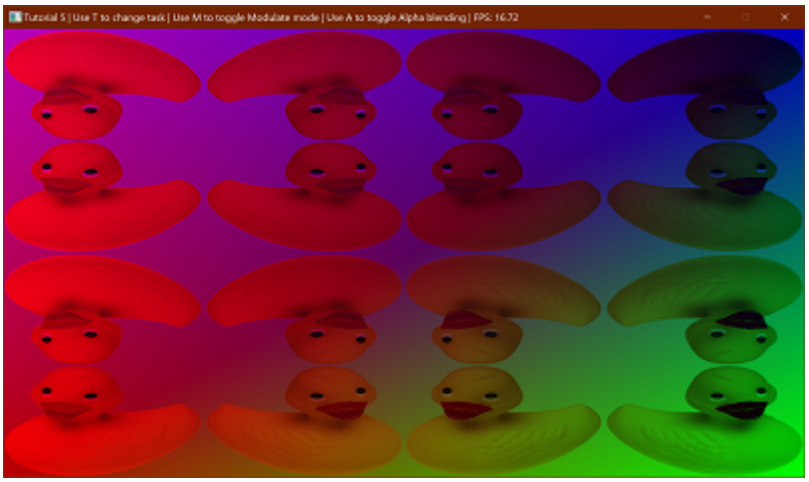

Figure 24: Repeating and mirroring texture mapping with modulated fragment color

Execute the sample to view the correct image that your program should generate when keyboard buttons $\mathbf{M}$ and $\mathbf{A}$ are separately toggled.

🎯Task 6: Clamping textures to edge

For this task, the application must do the following:

1. In the ***vertex shader*** [why not the fragment shader?], scale texture coordinates by a value of 4 before they are passed to the rasterizer.
2. Set the texture parameters to `GL_CLAMP_TO_EDGE` mode along horizontal and vertical directions.

Figure 25: Clamping texture coordinates to edge of image

Figure 25 illustrates clamping of an image to the edge - your picture must match the image in Figure 25.

The user can toggle the modulate mode by pressing the keyboard button $\mathbf{M}$. The image generated by your program when modulating colors should match the image in Figure 26.

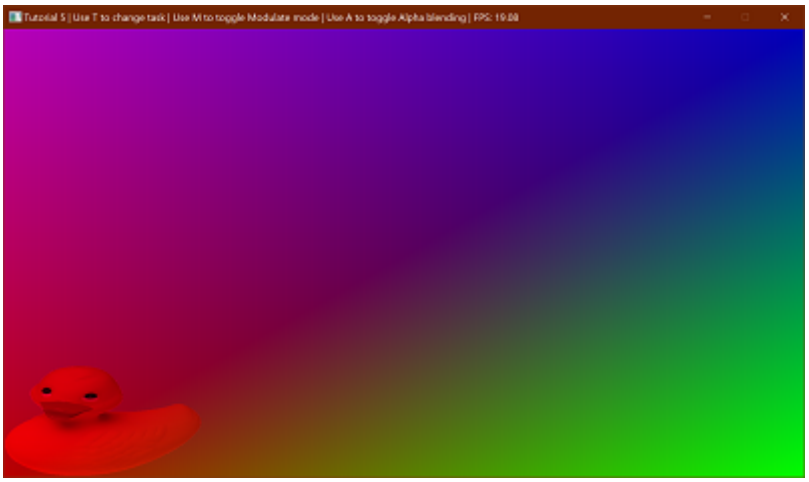

Figure 26: Clamping texture coordinates to edge of image and modulating texture and intepolated vertex colors

Execute the sample to view the correct image that your program should generate when keyboard buttons $\mathbf{M}$ and $\mathbf{A}$ are separately toggled.

🎯Task 7: Texture rotation

Begin this task by carefully studying the sample's behavior for tasks $3 - 6$ when keyboard button $\mathbf{R}$ is toggled. You'll notice the texture "rotating" about the mouse cursor while the rectangular polygon is static. Texture "rotation" is easily simulated by rotating the texture coordinates associated with each vertex of a triangular primitives or the texture coordinates associated with each fragment. There are three possibilities to implement texture "rotation":

- the application [that is, the CPU] rotates the per-vertex texture coordinates and updates the VBO with the updated texture coordinates
- the vertex shader [that is, the GPU] rotates the per-vertex texture coordinates and passes the rasterizer the updated texture coordinates
- the fragment shader [that is, the GPU] rotates the per-fragment texture coordinates and uses the updated per-fragment texture coordinates to sample the texture image.

There are two valid reasons for choosing the GPU over the CPU when processing vertices, fragments, and pixels.

- CPUs are SISD [Single instruction single data] processors that operate on a single element of data at a time. In contrast, GPUs are SIMD [Single instruction multiple data] processors that can operate on multiple sets of data simultaneously. To understand the difference between SISD and SIMD processors, consider the problem of computing the dot product of two vectors. Computing the dot product on a CPU requires 5 fundamental operations [3 multiplications and 2 additions]:

  ```
  1 struct my_vec3 {
  2   double x, y, z;
  3 };
  4
  5 my_vec3 p {1.0, 2.0, 3.0}, q {4.0, 5.0, 6.0};
  6 double  dot_prod {p.x*q.x + p.y*q.y + p.z*q.z};
  ```

  In contrast, the dot product computation is a single, atomic operation on a GPU with the fundamental operations occurring in parallel:

```
// dvec3 is GLSL vector type ...
dvec3  p = dvec3(1.0, 2.0, 3.0), q = dvec3(4.0, 5.0, 6.0);
double dot_prod = dot(p, q);
```

- CPUs have a small number of processing cores that are well suited for performing a wide variety of tasks. Each core concentrates on completing an individual task quickly. On the other hand, GPUs have hundreds or thousands of simpler and more specialized processing cores that can deliver high performance when they work together in parallel. For example, if the task is to rotate hundreds of per-vertex texture coordinates, each specific core can be assigned [by the graphics driver] the task of rotating the texture coordinate of a single vertex. The entire rotation task is completed simultaneously in parallel by all the cores.

Now that we know that we should use the GPU to process vertices and fragments, we must decide between using a vertex shader to transform per-vertex texture coordinates or a fragment shader to transform per-fragment texture coordinates. Recall that we're rendering a texture mapped rectangular polygon that exactly matches the viewport's shape and size. Further, the viewport maps to the entire display window. Such a rectangular polygon can be specified using two primitives of type `GL_TRIANGLES`. Each primitive requires no more than 3 vertices. Therefore, a vertex shader would only need to perform no more than 6 per-vertex texture coordinate rotations. How many transformations would a fragment shader have to perform to get the desired result? Given a viewport with dimensions 2400×1350, the fragment shader will process $2400 * 1350 = 3,240,000$ fragments. Thus, the fragment shader must perform $3,240,000$ per-fragment texture coordinate rotations. Comparing the number of rotations performed by the vertex and fragment shaders, the answer is clear: rotating 6 per-vertex texture coordinates in a vertex shader is far more optimal than rotating millions of per-fragment texture coordinates in a fragment shader.

Recall the 2×2 transformation matrix for rotating about $z-$axis through angle of θ radians [with respect to origin] is:

$$\mathrm{R}_\theta = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix}$$

Rotation matrix R_θ transforms texture coordinate (s, t) to a new texture coordinate (s', t'):

$$\begin{bmatrix} s' \\ t' \end{bmatrix} = \mathrm{R}_\theta \begin{bmatrix} s \\ t \end{bmatrix} = \begin{bmatrix} \cos(\theta) & -\sin(\theta) \\ \sin(\theta) & \cos(\theta) \end{bmatrix} \begin{bmatrix} s \\ t \end{bmatrix} = \begin{bmatrix} s * \cos(\theta) - t * \sin(\theta) \\ s * \sin(\theta) + t * \cos(\theta) \end{bmatrix}$$

GLFW function `glfwGetTime` returns the time elapsed in seconds [at the resolution of micro- or nanoseconds] since GLFW's initialization. Since $\theta \in [0, 2\pi)$, the increasing values returned by each call to `glfwGetTime` can be used as a suitable substitute for angle θ in rotation matrix R_θ.

One option in authoring the vertex shader is to provide angle θ as a uniform variable and let the vertex shader compute rotation matrix R_θ and transform the incoming per-vertex texture coordinate (s, t):

```
#version 450 core

layout (location = 0) in vec2 vVertexPosition;
layout (location = 1) in vec3 vVertexClrCoord;
layout (location = 2) in vec2 vVertexTexCoord;

layout (location = 0) out vec3 vClrCoord;
layout (location = 1) out vec2 vTexCoord;
```

```
9
10 uniform float uTheta;
11
12 void main () {
13   vClrCoord = vVertexClrCoord;
14
15   // rotate texture coordinates ...
16   float cos_theta = cos(uTheta), sin_theta = sin(uTheta);
17   vTexCoord = vec2(dot(vVertexTexCoord,vec2(cos_theta,-sin_theta)),
18                    dot(vVertexTexCoord, vec2(sin_theta, cos_theta)));
19
20   gl_Position = vec4(vVertexPosition, 0.f, 1.f);
21 }
```

Recall the vertex shader is executed once per vertex. For a rectangular polygon consisting of 4 vertices, the overall burden on the GPU is insignificant. However, when the vertex shader is applied on an object with thousands of vertices, the GPU's throughput will suffer. A better solution is to compute rotation matrix R_θ on the CPU and provide the 2×2 matrix as a uniform variable to the shader:

```
1 // compute rotation matrix ...
2 GLdouble theta {glfwGetTime()};
3 GLfloat cos_theta {static_cast<GLfloat>(std::cos(theta))},
4         sin_theta {static_cast<GLfloat>(std::sin(theta))};
5 glm::mat2 rot_mtx {cos_theta, sin_theta, -sin_theta, cos_theta};
6 // send rotation matrix as uniform variable to vertex shader ...
7 GLuint rot_mtx_loc = glGetUniformLocation(shdr_pgm.GetHandle(), "uRotMtx");
8 glUniformMatrix2fv(rot_mtx_loc, 1, GL_FALSE, glm::value_ptr(rot_mtx));
```

The vertex shader will now look like this:

```
1  #version 450 core
2
3  layout (location = 0) in vec2 vVertexPosition;
4  layout (location = 1) in vec3 vVertexClrCoord;
5  layout (location = 2) in vec2 vVertexTexCoord;
6
7  layout (location = 0) out vec3 vClrCoord;
8  layout (location = 1) out vec2 vTexCoord;
9
10 uniform mat2 uRotMtx;
11
12 void main () {
13   vClrCoord = vVertexClrCoord;
14
15   // rotate texture coordinates ...
16   vTexCoord = uRotMtx * vVertexTexCoord;
17
18   gl_Position = vec4(vVertexPosition, 0.f, 1.f);
19 }
```

Dragging the mouse cursor when executing the sample [after toggling keyboard button $\mathbf{R}$] in tasks 3 to 6 shows the texture rotated with respect to the mouse cursor. By default, texture coordinates are rotated in unit texture space $(s, t) \in [0, 1] \times [0, 1]$ with respect to origin $(0, 0)$ [located at bottom-left corner of texture space]. Suppose the display window has dimensions 2400×1350. GLFW returns mouse cursor coordinates $(x^m, y^m) \in [0, 2400) \times [0, 1350)$ with origin at top-left corner. Unfortunately, this coordinate system used by GLFW to record mouse cursor coordinates (x^m, y^m) doesn't match OpenGL's unit texture space [that is used to record texture coordinates supplied to the graphics pipe]. Mouse cursor coordinates (x^m, y^m) are transformed to unit texture space (s, t) like this:

$$\begin{aligned}&\text{Input: Display window's width and height are } W \text{ and } H, \text{ respectively}\\ &\text{Input: } (x^m, y^m) \text{ are mouse cursor coordinates returned by GLFW}\\ &\text{Output: } (x^m_s, y^m_s) \text{ are mouse cursor coordinates in unit texture space}\end{aligned}$$

$$x^m_s = \frac{x^m}{W}, \quad y^m_s = 1.0 - \frac{y^m}{H}$$

To rotate texture coordinates with respect to the mouse cursor, we need to first obtain mouse cursor coordinates. During GLFW's initialization [in source file glhelper.cpp], we had set up GLFW in function `GLHelper::setup_event_callbacks` with a callback function that returns mouse cursor's coordinates:

```
void GLHelper::setup_event_callbacks() {
  // other stuff ...
  glfwSetCursorPosCallback(GLHelper::ptr_window, GLHelper::mousepos_cb);
  // more stuff ...
}
```

The default implementation of the callback function will return the mouse cursor's current x and y coordinates in parameters `xpos` and `ypos`, respectively:

```
void GLHelper::mousepos_cb(GLFWwindow *pwin, double xpos, double ypos) {
#ifdef _DEBUG
  std::cout << "Mouse cursor position: (" << xpos << ", " << ypos << ")" <<
std::endl;
#endif
}
```

Amend structure `GLHelper` [in header file glhelper.h] to store normalized mouse coordinates:

```
struct GLHelper {
  // other previous stuff ...
  static glm::vec2 mcn;
}
```

Define static variable `mcn` in source file glhelper.cpp:

```
glm::vec2 GLHelper::mcn {0.f, 0.f};
```

and then amend function `mousepos_cb` to record the transformed mouse coordinates:

```
void GLHelper::mousepos_cb(GLFWwindow *pwin, double xpos, double ypos) {
  mcn.x = static_cast<float>(xpos / static_cast<double>(GLHelper::width));
  mcn.y = static_cast<float>(1.0 - ypos / static_cast<double>
(GLHelper::height));
}
```

Since the per-vertex texture coordinates are rotated in the vertex shader, the values recorded in `GLHelper::mcn` must be transferred to the vertex shader using an uniform variable:

```
// class GLSLShader has member function SetUniform that
// simplifies the transfer of GLHelper::mcn to a shader ...
shdr_pgm.SetUniform("uMcn", GLHelper::mcn);
```

The final step is to program the vertex shader to rotate the per-vertex texture coordinates with respect to the mouse cursor's normalized coordinates in uniform variable `uMcn`. This requires applying a *coordinate transform* [in contrast, to a *geometric transform*] to the per-vertex texture coordinates (s, t) so that they're defined relative to the mouse cursor's normalized coordinates `uMcn` [and not with respect to the bottom-left origin $(0, 0)$]:

$$\begin{bmatrix} s' \\ t' \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & -\text{uMcn.x} \\ 0 & 1 & -\text{uMcn.y} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} s \\ t \\ 1 \end{bmatrix} = \begin{bmatrix} s - \text{uMcn.x} \\ t - \text{uMcn.y} \\ 1 \end{bmatrix}$$

The updated vertex shader that satisfies the requirements for this task will now look like this:

```
#version 450 core

layout (location = 0) in vec2 vVertexPosition;
layout (location = 1) in vec3 vVertexClrCoord;
layout (location = 2) in vec2 vVertexTexCoord;

layout (location = 0) out vec3 vClrCoord;
layout (location = 1) out vec2 vTexCoord;

uniform vec2 uMcn;
uniform mat2 uRotMtx;

void main () {
  vClrCoord = vVertexClrCoord;

  // send rotated texture coordinates to subsequent stages ...
  vTexCoord = uRotMtx * (vVertexTexCoord - uMcn);

  gl_Position = vec4(vVertexPosition, 0.f, 1.f);
}
```

🎯Task 8: Printing to title bar

Print information in window toolbar *exactly* similar to sample executable.

Submission

1. Create a copy of project directory tutorial-5 named <login>-<tutorial-5>. That is, if your Moodle student login is foo, then the directory should be named foo-tutorial-5. Ensure that directory foo-tutorial-5 has the following layout:

```
foo-tutorial-5          # You're submitting Tutorial 5
├─ include              # Header files - *.hpp and *.h files
└─ src                  # Source files - *.cpp and .c files
└─ shaders              # Shader files - *.vert and .frag files
   └─ my-tutorial-5.vert # Vertex shader file
   └─ my-tutorial-5.frag # Fragment shader file
```

2. *Zip the directory* to create archive file foo-tutorial-5.zip.

⚠ Before Submission: Verify and Test ⚠

- Copy archive file foo-tutorial-5.zip into directory test-submissions. Unzip the archive file to create project directory foo-tutorial-5 by typing the following command in the command-line shell [which can be opened by typing cmd and pressing Enter in the Address Bar]:

```
powershell -Command "Expand-Archive -LiteralPath foo-tutorial-5.zip -DestinationPath ."
```

- After executing the command, the layout of directory test-submissions will look like this:

```
icg-opengl-dev    # Sandbox directory for all assessments
├─ test-submissions  # ⚠Test submissions here before uploading
|  └─ foo-tutorial-5 # foo is submitting Tutorial 5
|     └─ include     # Header files - *.hpp and *.h files
|     └─ src         # Source files - *.cpp and .c files
|     └─ shaders     # Shader files - *.vert and .frag files
|        └─ my-tutorial-5.vert  # Vertex shader file
|        └─ my-tutorial-5.frag  # Fragment shader file
└─ icg.bat              # Automation Script
```

- Delete the original copy of foo-tutorial-5 from directory projects to prevent duplicate project names during reconfiguration.
- Run batch file icg.bat and select option R to reconfigure the Visual Studio 2022 solution with the new project foo-tutorial-5.
- Build and execute project foo-tutorial-5 by opening the Visual Studio 2022 solution in directory build.
- Use the following checklist to ***verify and submit*** your submission:

Things to test before submission	Status
Assessment compiles without any errors	☐
All compilation warnings are resolved; there are zero warnings	☐

Things to test before submission	Status
Executable generated and successfully launched in Debug and Release mode	☐
Directory is zipped using the naming conventions outlined in submission guidelines	☐
Zipped file is uploaded to assessment submission page	☐

The purpose of this verification step is not to guarantee the correctness of your submission. Instead, it verifies whether your submission meets the most basic rubrics [it compiles; it doesn't generate warnings; it links; it executes] or not and thus helps you avoid major grade deductions. And, remember you'll need every single point from these programming assessments to ensure a noteworthy grade!!! Read the section on Grading Rubrics for information on how your submission will be assigned grades.

Grading Rubrics

The core competencies assessed for this assessment are:

- [***core1***] Submitted code must build an executable file with ***zero*** warnings. This rubric is not satisfied if the generated executable crashes or displays nothing. In other words, if your source code doesn't compile nor link nor execute, there is nothing to grade.
- [***core2***] This competency indicates whether you're striving to be a professional software engineer, that is, are you implementing software that satisfies project requirements [are you submitting the required files? are they properly named? are they properly archived? are you submitting unnecessary files?], that is maintainable and easy to debug by you and others [are file and function headers properly annotated?]. Submitted source code must satisfy ***all*** requirements listed below. Any missing requirement will decrease your grade by one letter grade.
    - Source code must compile with ***zero*** warnings. Pay attention to all warnings generated by the compiler and fix them.
    - Source code files submitted is correctly named.
    - Source code files are *reasonably* structured into functions and *reasonably* commented. See next two points for more details.
    - If you've created a new source code file, it must have file and function header comments.
    - If you've edited a source code file provided by the instructor or from a previous tutorial, the file header must be annotated to indicate your co-authorship and the changes made to the original or previous document. Similarly, if you're amending a previously defined function, you must annotate the function header to document your amendments. You must add a function header to every new function that you've defined.
    - Keyboard button T allows me to cycle through and test individual tasks. If this feature is not implemented or implemented incorrectly, I can only grade one of the seven tasks.
- [***core3***] Task 0 is complete.
- [***core4***] Task 1 is complete.
- [***core5***] Task 2 is complete.

- [***core6***] Task 3 is complete including the toggling of color modulation and alpha blending through keyboard buttons $\mathbf{M}$ and $\mathbf{A}$, respectively.
- [***core7***] Task 4 is complete including the toggling of color modulation and alpha blending through keyboard buttons $\mathbf{M}$ and $\mathbf{A}$, respectively.
- [***core8***] Task 5 is complete including the toggling of color modulation and alpha blending through keyboard buttons $\mathbf{M}$ and $\mathbf{A}$, respectively.
- [***core9***] Task 6 Clamp-to-edge function is complete including the toggling of color modulation and alpha blending through keyboard buttons $\mathbf{M}$ and $\mathbf{A}$, respectively.
- [***core10***] Task 7 Texture "rotation" [toggled by keyboard button $\mathbf{R}$] is complete and is exactly similar to the sample.
- [***core11***] Task 8 Print information in window toolbar *exactly* similar to sample executable.

Mapping of Grading Rubrics to Letter Grades

The following table illustrates the mapping of core competencies listed in the grading rubrics to letter grades:

Grading Rubric Assessment	**Letter Grade**
There is no submission.	F
core1 rubric is not satisfied. Submitted property page and/or source code doesn't build. Or, executable generated by build crashes or displays nothing.	F
If ***core2*** rubrics are not satisfied, final letter grade will be decreased by one. This means that if you had received a grade A and ***core2*** is not satisfied, your grade will be recorded as B, an $A-$ would be recorded as $B-$, and so on.	
One of ***core3*** thro' ***core11*** rubrics are completed.	D
Two of ***core3*** thro' ***core11*** rubrics are completed.	$D+$
Three of ***core3*** thro' ***core11*** rubrics are completed.	C
Four of ***core3*** thro' ***core11*** rubrics are completed.	$C+$
Five of ***core3*** thro' ***core11*** rubrics are completed.	$B-$
Six of ***core3*** thro' ***core11*** rubrics are completed.	B
Seven of ***core3*** thro' ***core11*** rubrics are completed.	$B+$
Eight of ***core3*** thro' ***core11*** rubrics are completed.	A
All nine of ***core3*** thro' ***core11*** rubrics are completed.	$A+$