Lab 7: Intro to Vertex Shaders & Transformations

Publish | Edit | ⋮

GPR-200: Introduction to Modern Graphics Programming
Instructor: Daniel S. Buckstein
Lab 7: Intro to Vertex Shaders & Transformations

Summary:
In the course so far, we have strictly explored GLSL fragment shaders using the online editor Shadertoy. Here we were introduced to a variety of intro-level techniques and practices in shader programming, including applications such as fundamental lighting and post-processing. In this lab, we extend our knowledge of shader programming into ***vertex shaders***, which we use to process ***vertices***, the building blocks of 3D mesh-based models and geometry. We will explore another web-based editor capable of modifying both vertex and fragment shaders, the principles of both, and revisit some of the things we were able to do with only fragments, now using 3D geometry.

Submission:
Start your work immediately by ensuring your coursework repository is set up, and public. Create a new main branch for this assignment. ***Please work in pairs (see team sign-up) and submit the following once as a team***:

1. Names of contributors
   e.g. **Dan Buckstein**
2. A link to your public repository online
   e.g. **https://github.com/dbucksteinccorg/graphics2-coursework.git**
   (note: this not a real link)
3. The name of the branch that will hold the completed assignment
   e.g. **lab0-main**

4. A link to your video (see below) that can be viewed in a web browser.
   e.g. <insert link to video on YouTube, Google Drive, etc.>

Finally, please submit a ***5-minute max*** demo video of your project. Use the screen and audio capture software of your choice, e.g. Google Meet, to capture a demo of your project as if it were in-

class. This should include at least the following, in enough detail to give a thorough idea of what you have created (hint: this is something you could potentially send to an employer so don't minimize it and show off your professionalism):

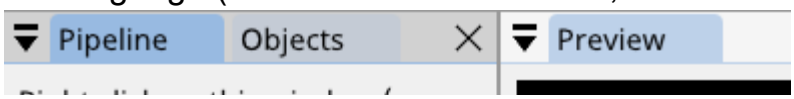
- Show the final result of the project and any features implemented, with a voice over explaining what the user is doing.
- Show and explain any relevant contributions implemented in code and explain their purpose in the context of the assignment and course.
- Show and explain any systems source code implemented, i.e. in framework or application, and explain the purpose of the systems; this includes changes to existing source.
- **DO NOT AIM FOR PERFECTION, JUST GET THE POINT ACROSS**. Please mind the assignment rubric to make sure you have demonstrated enough to cover each category.
- **Please submit a link to a video visible in a web browser, e.g. YouTube or Google Drive.**

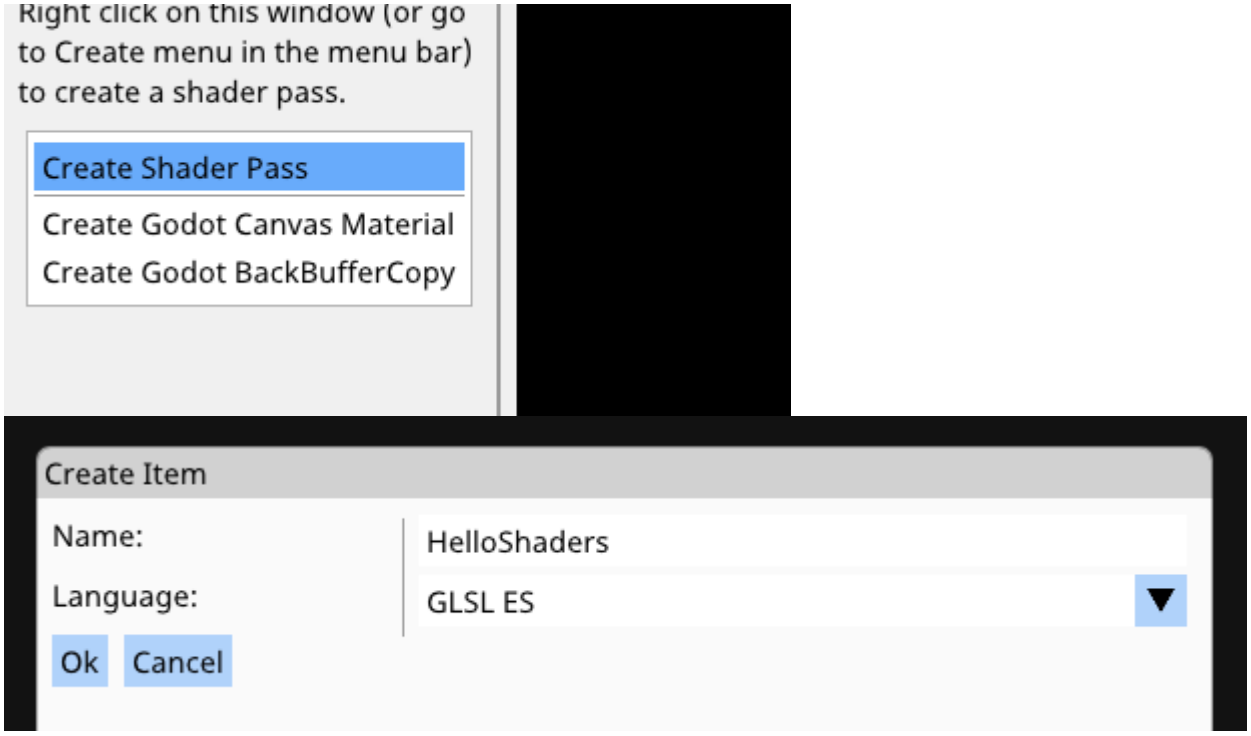
Objectives:
The purpose of this assignment is to introduce you to vertex shaders and expand your knowledge of the programmable graphics pipeline. You will revisit some lighting and texturing principles, as seen in lab 4 and lab 5, and implement them using a variety of 3D transformations and spaces on vertex-based models.

Instructions & Requirements:
DO NOT begin programming until you have read through the complete instructions, bonus opportunities and standards, start to finish. Take notes and identify questions during this time. The only exception to this is whatever we do in class.
Review the following prerequisite information, ***which contains important terms and vocabulary***, then implement the specified algorithms below:

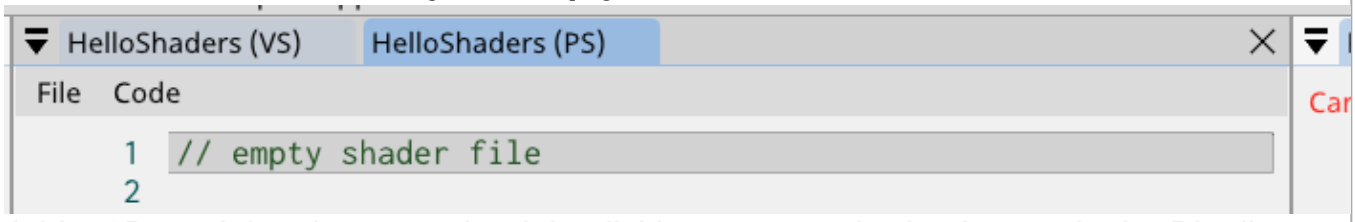
1. ***Setup and intro to editor***: For this assignment, we will use an online shader editor called **SHADERed** (https://shadered.org/). For class we will use the online (**lite** (https://shadered.org/template)) version, based in WebGL (and GLSL ES), but you are welcome to explore the desktop version, based in cross-platform OpenGL (and GLSL), on your own.
   - Go to **SHADERed** (https://shadered.org/) and create an account. This will allow you to save and revisit your work.
   - For class, we will be using the online version (**lite** (https://shadered.org/template)), which has a number of template projects with which to begin coding your shaders. We will start with an **empty project** (https://shadered.org/app) (top-left template).
   - When the editor opens, you will see a blank preview window and some options on the left.

   - To begin coding, right click in the Pipeline menu in the left panel and select "Create Shader Pass". In the menu that opens, give your "shader pass" a name and choose the language (either GLSL or GLSL ES; the differences will be minimal, if any).

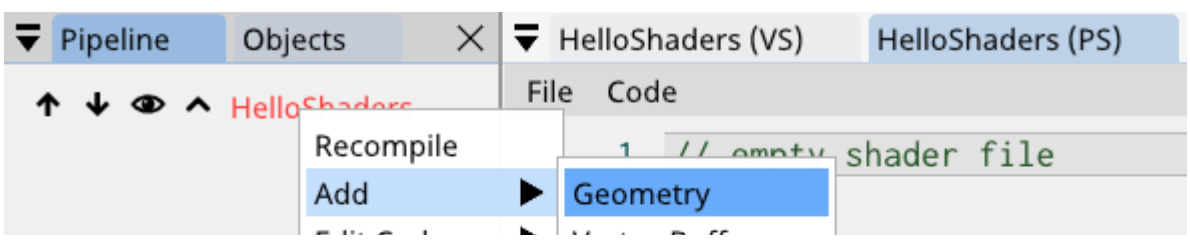


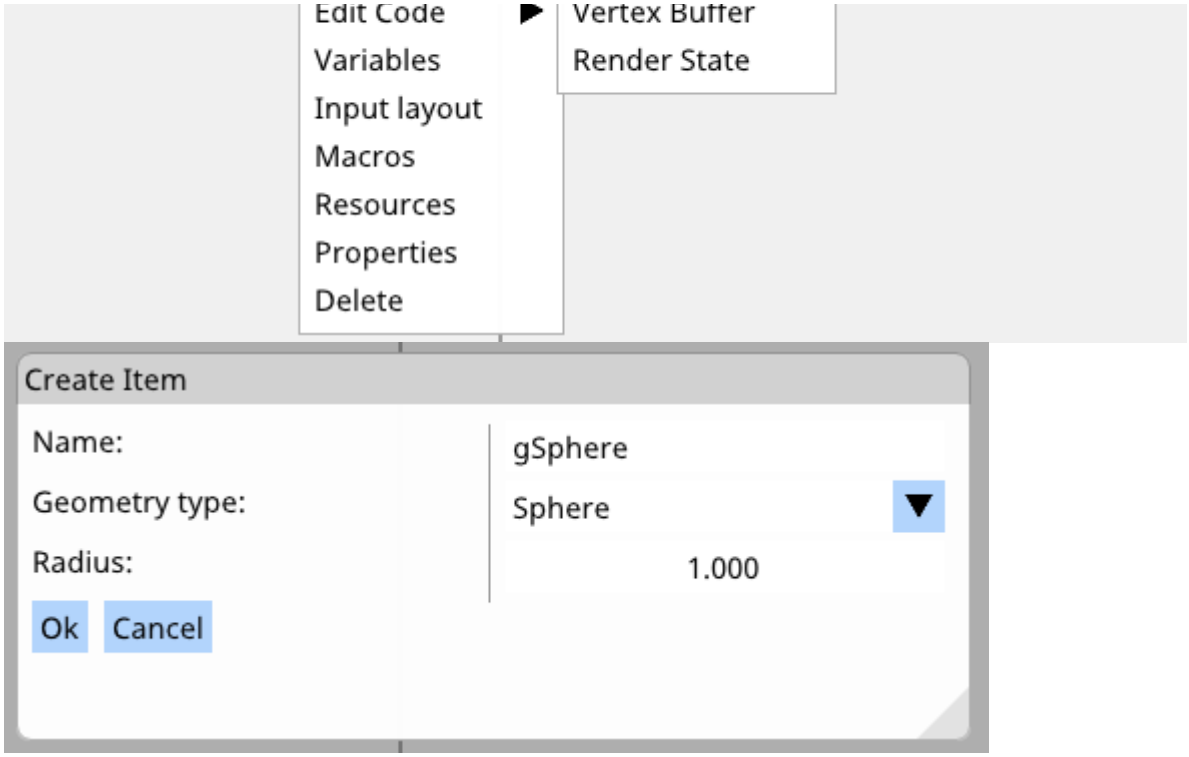


- Upon creation, your "shader pass" will appear as a pair of shaders at the top of the window: "<previously selected name> (VS)" (the ***vertex shader***) and "<previously selected name> (PS)" (the ***fragment shader***, we'll discuss why it says 'PS' below). The name you selected will appear in the pipeline tab on the left. ***Note: the shaders will not build because they are empty; we will write them soon!***



- Add a 3D model to the scene by right clicking on your shaders' name in the Pipeline panel, select "Add > Geometry" and the shape of your choice (e.g. a sphere). Give it a name, configure its settings and confirm. It will appear under the shaders' name in the pipeline panel.





- Before we write the shaders, set up your version control pipeline and make sure you save your work. For version control, commit frequently to your repository by copying the contents of each shader file to a text file (e.g. "vertex.glsl" and "fragment.glsl").
- You can save and re-access your SHADERed project a couple of different ways:
  - To save to your account, click the "Upload" button next to your user name on the top left corner of the page. Give it a name and description and upload. ***Uncheck the 'public' button to keep it private***.
  - To reopen, exit the editor and click on your username in the top-right corner. Here you can view your projects. Click on one to access a "quick edit" interface, much like Shadertoy's, and click "Fork" below the preview window to continue work on the project.
  - Alternatively, you can download the project for use in the desktop version. Go to "File > Download".
  - ***Note: Like Shadertoy and any other tool, SHADERed may crash or bug out unexpectedly; please keep a running backup of your work in text documents.***

2. ***Shaders and programs***: Before we proceed into shader land, we must go over the basics; detailed descriptions may be found in the books and specifications. Keep an eye out for the ***bolded terms*** throughout this briefing.
   - **Shader**: It is a slight misnomer as it doesn't necessarily "shade"; a ***shader*** simply refers to any programmable stage in the graphics pipeline. OpenGL has ***five*** programmable stages in the vertex pipeline, with ***one*** independent programmable unit. In this course, we are concerned with two types:
     - ***Vertex shader***: The first stage in the programmable pipeline (and also the only required stage), the ***vertex shader*** has two primary responsibilities:
       - Read ***vertex attributes*** (see data categories below) from a vertex buffer

prepared by the engine/application.
        - Operate on a ***single vertex***, transforming its position into ***clip-space*** for primitive assembly later in the pipeline.
    - ***Fragment shader***: The final stage in the programmable pipeline, the ***fragment shader*** has one primary responsibility:
        - Operate on a ***single fragment***, using its properties to calculate and write a ***pixel color*** to the active ***framebuffer***.
        - *The fragment shader is often colloquially referred to as a "pixel shader" (hence, 'PS' in SHADERed), but it is important to note that a fragment is not the same as a pixel (more details below).*
    - There are other types of shaders that we are not concerned with at this time, but they are worth reading about: *tessellation control/evaluation*, *geometry* and *compute*.
- **Shader program**: What we have previously referred to as a "shader pass", as it is called in SHADERed, is actually called a ***shader program***: the complete programmable pipeline that begins with a ***vertex shader*** (required) to read vertex attributes, and ends with a ***fragment shader*** to write to the target framebuffer.
    - The rendering pipeline consists of the five shader stages in whatever program is active for rendering: vertex, tessellation control, tessellation evaluation, geometry and fragment. Programs may also include a compute shader which operates independently.

3. ***Data categories***: Here is your quick reference card for what kinds of ***data*** you're dealing with and what it all means. Remember: the GPU doesn't know what you're feeding it, so it is your responsibility to manage the "just data" that it sees! Complete details in the books and specifications. The raw accessible data categories are summarized here in **bold**, with related key terms in *italic*:
    - **Attribute**: A ***vertex attribute*** is any descriptive component of a vertex used to build a 3D model or mesh. These are things that can be different for each vertex in a mesh; e.g. position, normal, etc.
        - Note: Attributes are accessible in the ***vertex shader only*** and are the shader's ***inputs***. They are ***read-only***. We will learn how to configure them using SHADERed in class.
        - Declaration: Attributes are declared using the '*in*' qualifier. In older versions of GLSL, they used the 'attribute' keyword instead. The OpenGL standard allows a maximum of 16 attributes. Here are a couple examples using modern techniques:

```
// EXAMPLE 1: Data is prepared in application, which plans for this attribute to be
//              linked corresponding to a specific location; name doesn't matter.
layout (location = 0) in vec3 aPosition;

// EXAMPLE 2: Shader decides location at link time based on name chosen by application;
//              must use the same name chosen here.
in vec3 normal;
```

        - Related terms:
            - *Vertex*: A collection of attributes describing a discrete sample on a model. The

vertex shader is responsible for reading the attributes of a single vertex at a time and using those to create primitives.
        - *Primitive*: A collection of vertices forming some basic 3D shape, such as points (1 vertex each), line segments (2 vertices each) and triangles (3 vertices each); there are more primitive types, but they all boil down to these three.
        - *Primitive assembly*: A non-programmable stage in the rendering pipeline during which processed vertices are "grouped" to represent your desired primitive.
        - *Model*: A collection of primitives forming a complex 3D mesh or geometric object, such as a sphere or plane. Pro-tip: seeing an image on the display only works because we're drawing a plane!
- **Render target**: A ***render target*** is the destination for pixel colors, such as a texture or the display buffer itself.
    - Note: Render targets are accessible in the ***fragment shader only*** and are the shader's ***outputs***.
    - Declaration: Render targets are declared using the '*out*' qualifier. In older versions of GLSL, they were not used, instead represented by built-in GLSL variables. The number of targets depends on the card manufacturer. Here are a couple examples using modern techniques:

```
// EXAMPLE 1: Targeting a specific "layer" in the active framebuffer; name doesn't matter.
// ***NOTE: THESE LOCATIONS ARE ENTIRELY INDEPENDENT FROM THE VERTEX ATTRIBUTE LOCATIONS***
layout (location = 0) out vec4 rtFragColor;
// EXAMPLE 2: If no location is specified, targets default buffer; name doesn't matter.
out vec4 finalCol;
```

    - Related terms:
        - *Fragment*: After primitives are generated, they are then *rasterized*, which breaks them down into tiny boxes known as *fragments*. These are not to be confused with pixels because they still represent a piece of a 3D primitive and are not part of an image yet. The fragment shader operates on a single fragment at a time and generates a color to be written to the target buffer.
        - *Rasterization*: The process by which a 3D shape is converted to fragments, the smallest possible representation of displayable in a 3D scene.
        - *Pixel*: The output of a fragment shader is a color; this is stored in the target framebuffer as a pixel in a 2D image.
        - *Framebuffer*: Stores the final rendered image; each invocation of the fragment shader passes a color to be written at the corresponding pixel's location in the framebuffer.

- **Varying**: A ***varying*** is some variable that transfers data down the pipeline to subsequent shader stages. In this assignment, we use varyings to pass data from the vertex shader to the fragment shader.
    - Note: Varyings exist in all shader types. In stages closer to the beginning of the pipeline, they are the ***outputs*** and in later stages are the ***inputs***. E.g. if passing

data from a vertex shader to a fragment shader, the varying is an ***output*** of the vertex shader and an ***input*** of the fragment shader. Any such variable must have the ***same name in both places***. In a fragment shader, varyings are interpolated by the rasterizer to "fill the space" between vertices.
    - Declaration: Varyings exist at all stages in the pipeline, using the '*out*' qualifier in earlier stages (must be written), and the '*in*' qualifier in later stages (read-only). In older versions of GLSL, they used the 'varying' keyword. In any case, matching varyings must have the same data type and name; the only part that changes is the in/out qualifier:

```
// EXAMPLE: Data to be passed from a vertex shader to a fragment shader.
//--------------------------------
// OUTPUT from vertex shader (VS writes value):
out float vSomeScalarVS2FS;
//--------------------------------
// INPUT to fragment shader (FS reads value, must have same name and type):
in float vSomeScalarVS2FS;
//--------------------------------
```

- **Uniform**: A ***uniform*** is some variable that retains the same value for an entire draw call and is sent from the application.
    - Note: Uniform variables are accessible to ***all shaders*** in the program (e.g. both vertex and fragment) and has the ***exact same value*** or configuration for ***all invocations of all shaders*** until changed for the the next draw call. E.g. if the model is just a triangle, a uniform would be set before the triangle is drawn, and would be the same for all three vertices and however many fragments are generated.
    - Declaration: Uniforms exist at all stages in the pipeline, using the '*uniform*' qualifier. Uniforms are accessible to all shaders in a program and are read-only:

```
// EXAMPLE: Some uniform in a program.
//--------------------------------
// Some texture in the vertex shader:
uniform sampler2D uTexture;
//--------------------------------
// Some texture in the fragment shader (exactly the same):
uniform sampler2D uTexture;
//--------------------------------
```

4. ***Transformations and spaces***: Aside from the above vocabulary, we must also have some understanding of ***vertex transformations and spaces***. Here is a list of commonly used spaces in the transformation pipeline, and the transformation matrices used to get between spaces:
    - **Object-space**: A point or vector in ***object-space*** is relative to the object or model being rendered. Typically, raw attribute data is sourced in object-space.
    - **World/scene-space**: A point or vector in ***world-space*** is relative to the world or scene; it is the common space in which all objects exist.
        - Transformation from object- to world-space is done using the object's ***model matrix***. In SHADERed, this is built-in, referred to as the "geometry transform".
        - Transformation from world- to object-space is done using the inverse of the

respective model matrix.
    - **View/camera-space**: A point or vector in ***view-space*** is relative to the camera or point-of-view being used to draw the scene; it is as if the world is relative to the observer.
        - Transformation from world- to view-space is done using the camera's ***view matrix***, which is the inverse of its own model matrix (per the second transformation rule above). The view matrix is built-in with SHADERed.
        - Transformation from object- to view-space is done using the object's ***model-view matrix***, which is the concatenation of its model matrix and the camera's view matrix.
        - An alternative version of an object's model-view matrix exists for normals: the ***normal matrix*** is the inverse-transpose of the object's model-view matrix.
        - View-space is the last chance for forward shading, such as the Lambert and Phong algorithms we have discussed.
    - **Clip-space**: This is a non-linear space used for ***primitive clipping***, and the only real requirement of any shader program is that the vertex shader sets the vertex position in ***clip-space***.
        - Transformation from view- to clip-space is done using the camera's ***projection matrix***. The projection matrix is built-in with SHADERed, either as "projection" for a standard perspective projection, or "orthographic" for a parallel projection. There are also pre-concatenated versions of these which form the "view-projection" matrix, representing a transformation from world-space to clip-space.
        - Transformation from object- to clip-space is done using the object's ***model-view-projection matrix***, a concatenation of all three.
    - There are other spaces that we are not concerned with at this time, but they are worth reading about: *normalized device coordinates (NDC)* and *screen-space*.
5. ***Getting started***: Back to the shaders generated in SHADERed, add the following:
    - Start by adding the GLSL version; this must be the very first thing in each file. The version may be changed at any time; note that the language does change a bit based on the version.
        - If using GLSL, use "*#version 330*" or higher (up to 450).
        - If using GLSL ES, there are exactly two options for WebGL: "*#version 100 es*" for WebGL1 and "*#version 300 es*" for WebGL2. The latest version for other ES platforms, such as mobile, is "*#version 320 es*".
    - If using GLSL ES (WebGL or mobile), the fragment shader must also take the additional step of defining the precision of floats:

        - Use "*precision highp float;*" for the highest precision available; the other options are '*mediump*' and '*lowp*'.
        - To switch between GLSL and GLSL ES easily with one less thing to worry about (e.g. programming for cross-platform), wrap the precision setting in a macro, like this:

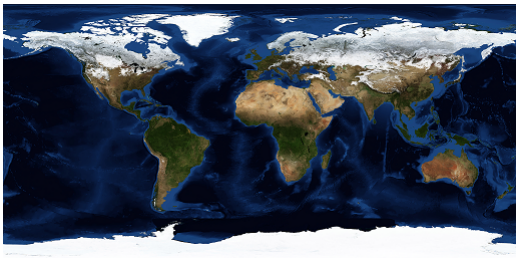
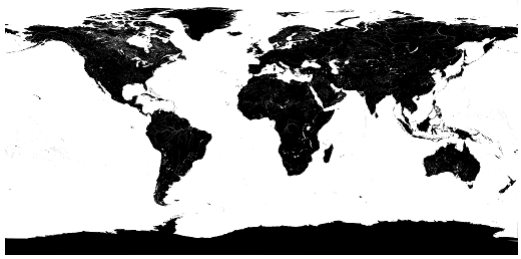
```
#ifdef GL_ES
precision highp float; // If GLSL ES is detected, add required precision setting.
#endif // GL_ES
```

- Finally, add your main function in each shader. This is not the same as Shadertoy's '*mainImage*'; this is actual shader programming, where the entry function is just '*main*'.
  - Each shader must have a void-returning, no-parameter function called main: "*void main()*", like this:

```
void main()
{
    // SHADER LOGIC BEGINS HERE!
}
```

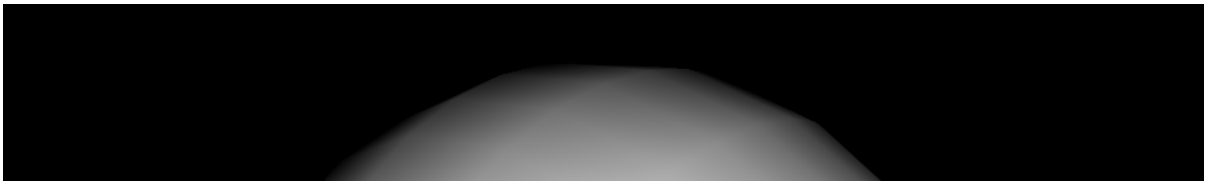
  - In the vertex shader, you must write a value to the built-in variable '*gl_Position*' by the end of main.
- You are now ready to proceed to the assignment, which will require you to declare the necessary attributes, targets, varyings and uniforms! All of the above has been the primer, some of which we will experience in class, but the main instructions and requirements of the assignment follow here.

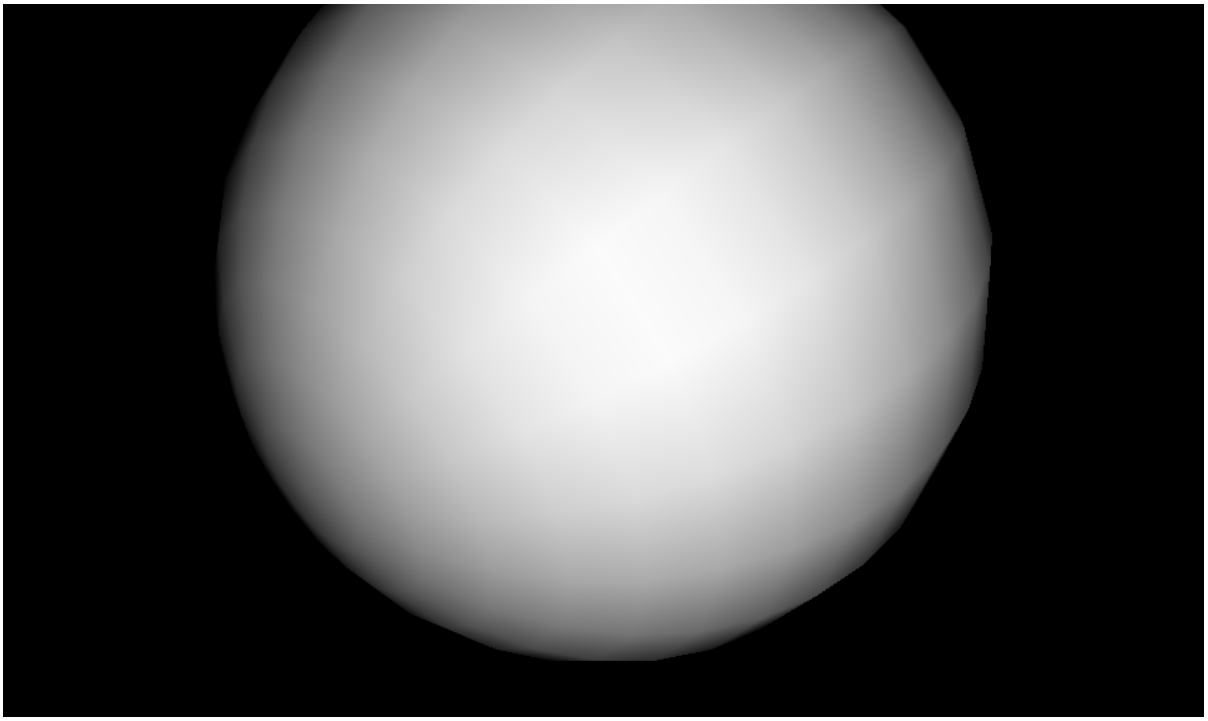
6. ***Phong shading on a 3D model***: For this assignment, implement several variations of the Phong shading algorithm, using a GLSL shader program consisting of a vertex and fragment shader. The goal of this is to experience the data flow between a vertex and fragment shader by focusing on a single task implemented in different ways. Here are the specifications for the assignment:
   - ***Constraints***: All of your Phong variations must meet the following constraints:
     - **Multiple lights**: Light the 3D object using at least ***three point lights***. Point light data may be defined using global variables, much like what we did in lab 4. All light positions must be defined relative to ***world-space***, or relative to the scene.
     - **Textures**: Use at least ***one texture*** for the surface color, as explored in lab 5. You may use separate textures for diffuse (surface) and specular (reflective) color, e.g. these earth textures (except please find some that are higher resolution):

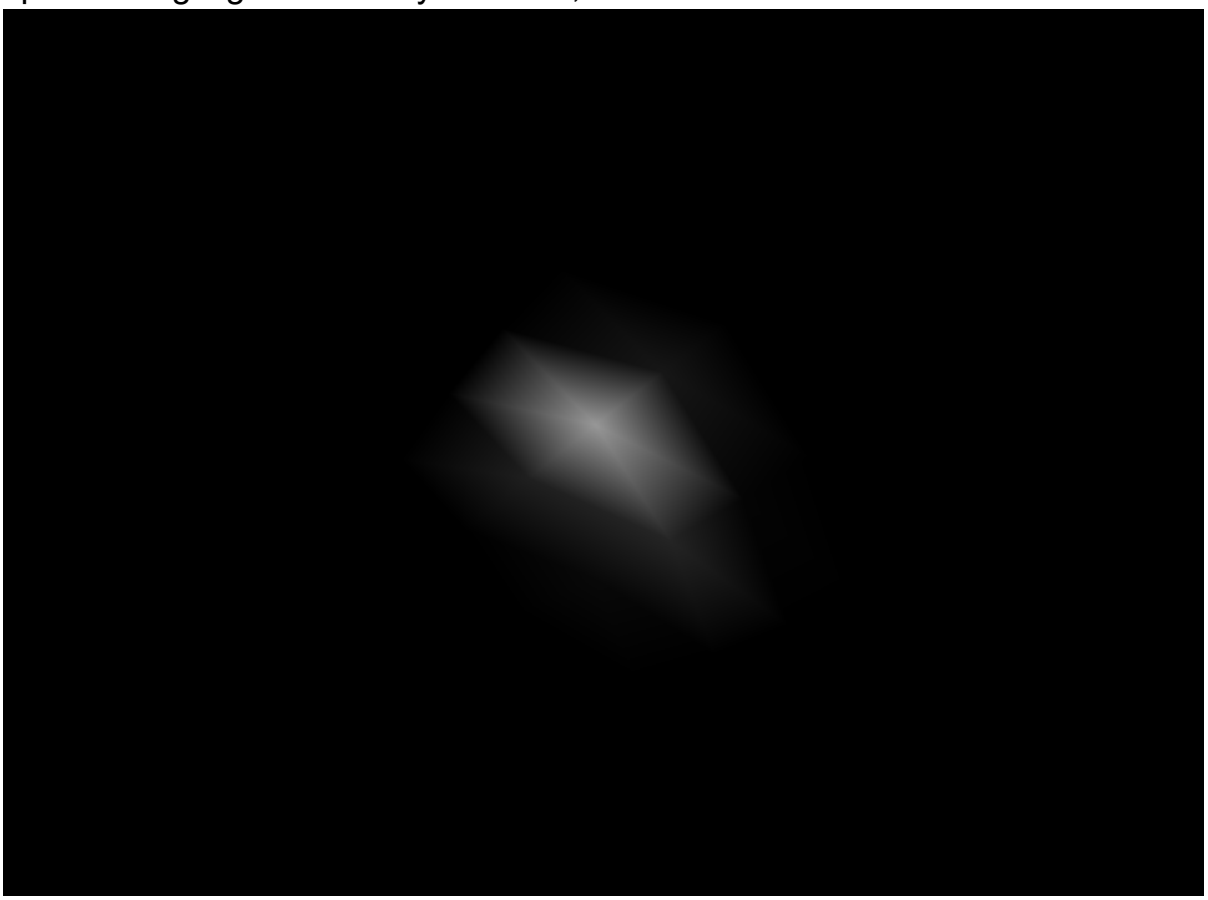
   - ***Variations***: Implement the following variations of Phong, leveraging the vertex and fragment shaders as needed. We will talk about the implications and details of the following in class:
     - **Per-vertex, view-space**:
       - This mode explores the faster yet lower-quality "***per-vertex shading***" in which the shading algorithm is entirely ***prepared and executed in the vertex shader***. The final output shows "blocky" lighting due to the relatively low

vertex concentration (i.e. there are generally far fewer vertices than fragments).
        - In the ***vertex shader***, all pertinent lighting variables (vertex position, light positions, normal) are transformed to ***view-space***, or relative to the camera, and are used here for lighting calculations.
        - The final required output of the vertex shader, '*gl_Position*', is always in ***clip-space***.
        - The final color is passed from the vertex shader to the fragment shader using a ***varying***.
        - The fragment shader simply receives the final color calculated in the vertex shader through a ***varying*** and displays it.
    - **Per-vertex, object-space**:
        - Same as above, except all pertinent lighting variables are transformed to ***object-space***, or relative to the 3D model itself, for lighting calculations.
        - The fragment shader does not change.
    - **Per-fragment, view-space**:
        - This mode explores the more expensive yet higher-quality (by far) "***per-fragment shading***" in which the shading algorithm data is ***prepared in the vertex shader***, and the algorithm itself is ***executed in the fragment shader***.
        - In the ***vertex shader***, all pertinent lighting variables are transformed to ***view-space***.
        - The final output of the vertex shader, '*gl_Position*', is always in ***clip-space***.
        - The view-space lighting variables are passed to the fragment shader using ***varyings***.
        - The fragment shader receives the pertinent lighting data from the vertex shader through ***varyings*** and uses it to perform the desired lighting/shading algorithm.
    - **Per-fragment, object-space**:
        - Same as above, except all pertinent lighting variables are transformed to ***object-space***, or relative to the 3D model itself, before being passed along to the fragment shader.
        - The fragment shader may change depending on your vertex shader implementation, but it is possible to have it stay the same.
- ***Gallery***: Here are some example images:
    - **Diffuse coefficient visualized with *per-vertex* shading**. Notice the subtle "banding" effect along the polygon edges. This happens because lighting is only calculated at the vertex locations, and the final color, the *output* of the lighting algorithm, is interpolated between vertices.

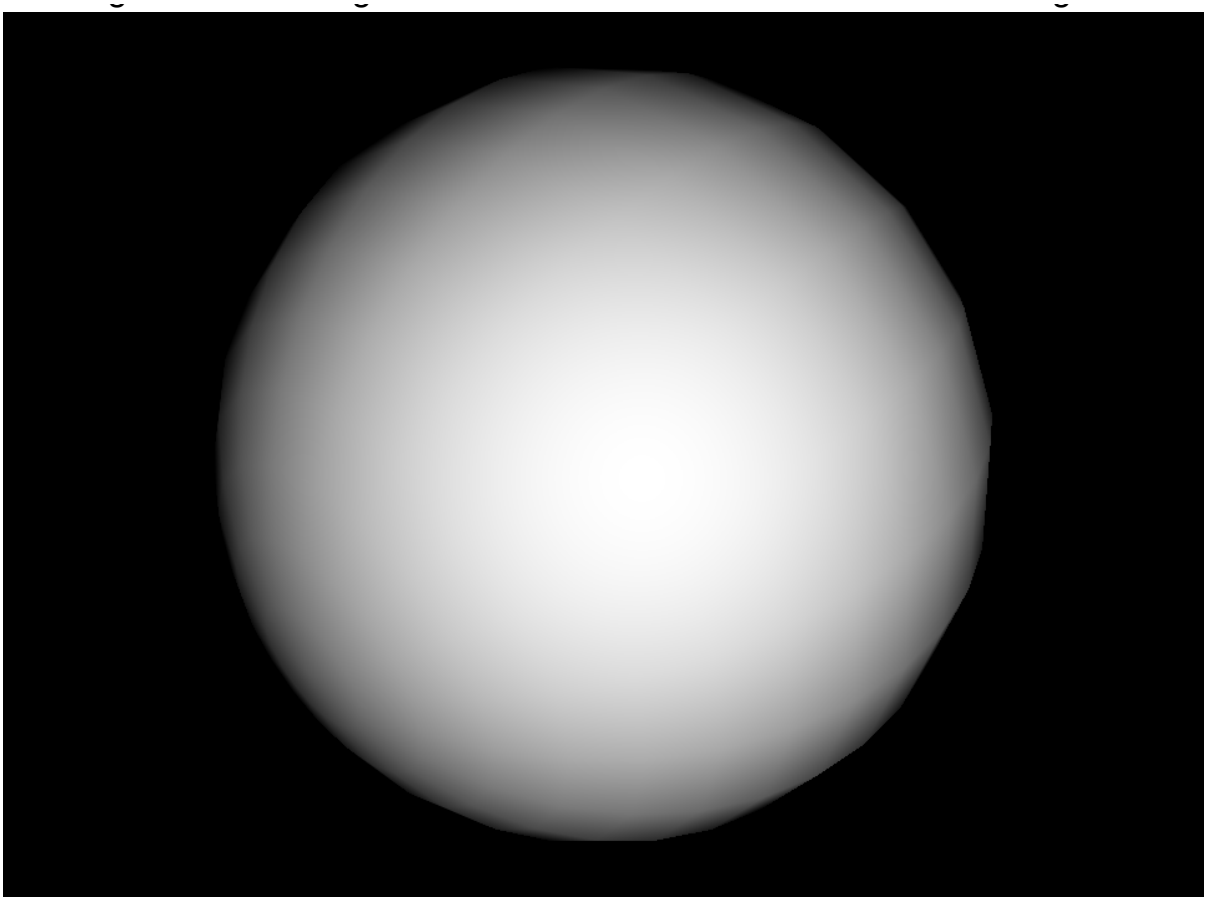




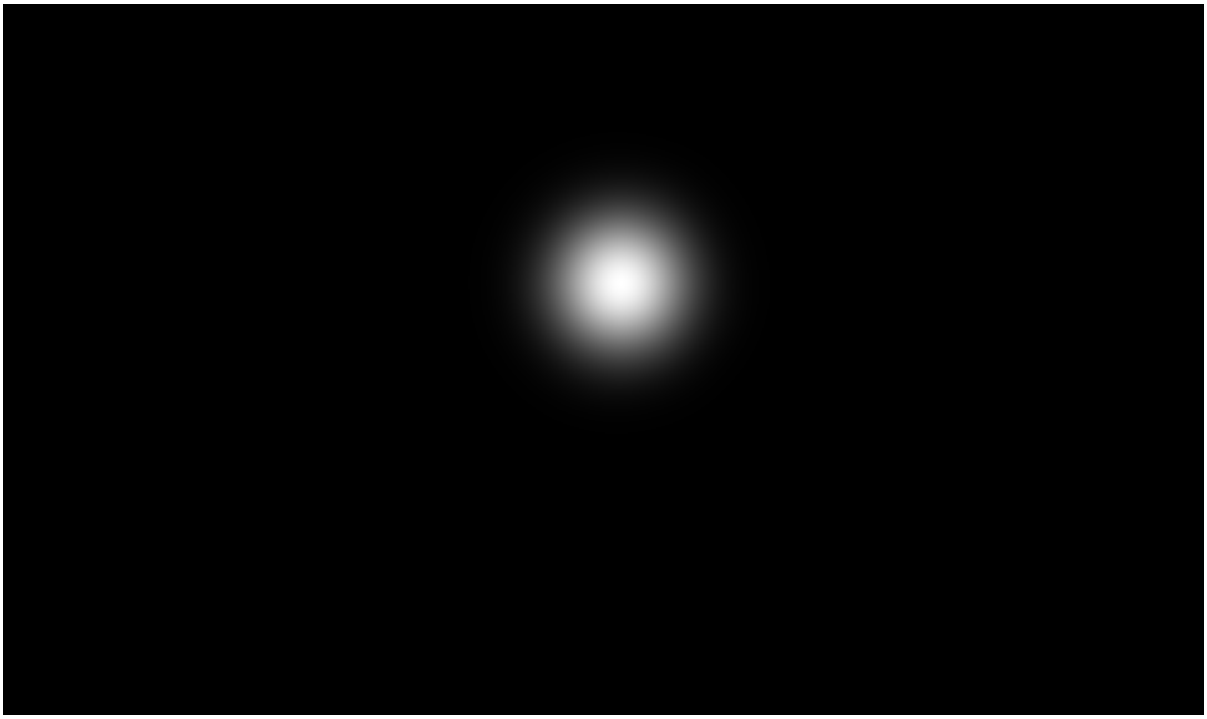
- **Specular highlight (power 16) visualized with *per-vertex* shading**. The banding is extremely apparent, again due to the low vertex resolution; since the specular highlight is already so small, this is the result.



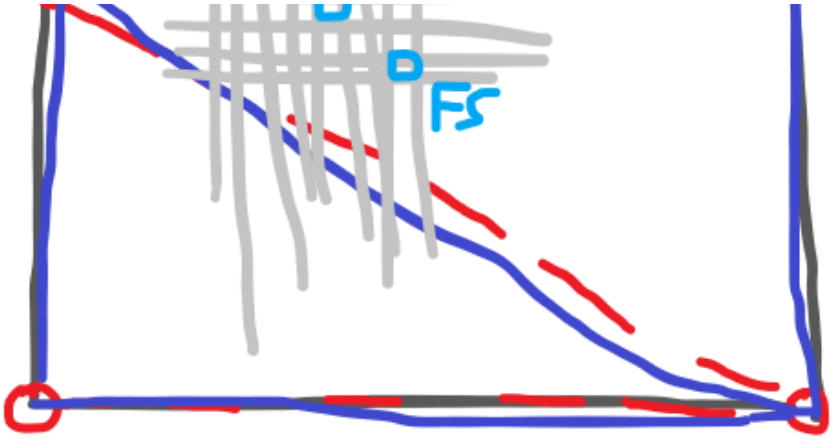
- **Diffuse coefficient visualized with *per-fragment* shading**. This image is much smoother because the *inputs* to the lighting algorithm are interpolated for each fragment, before actually being used to calculate lighting. There are many more fragments than vertices, so we have a higher resolution. There is still very subtle banding around the edges where we have a low concentration of fragments.

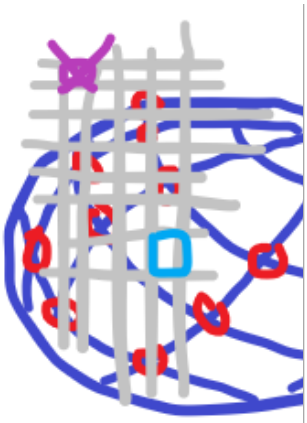


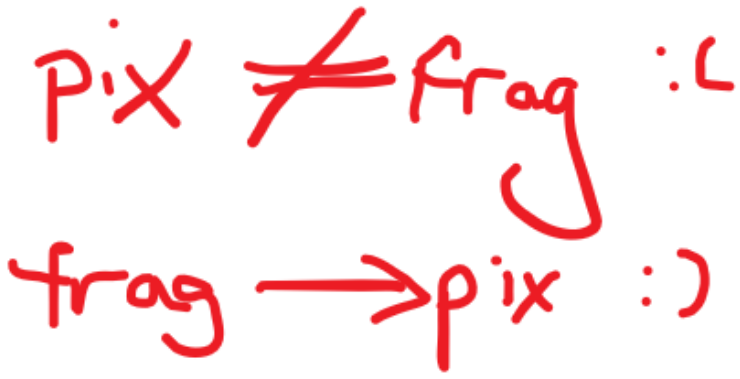
- **Specular highlight visualized with *per-fragment* shading**. The highlight is much smoother because the area is no longer constrained to being calculated only at the vertex locations.



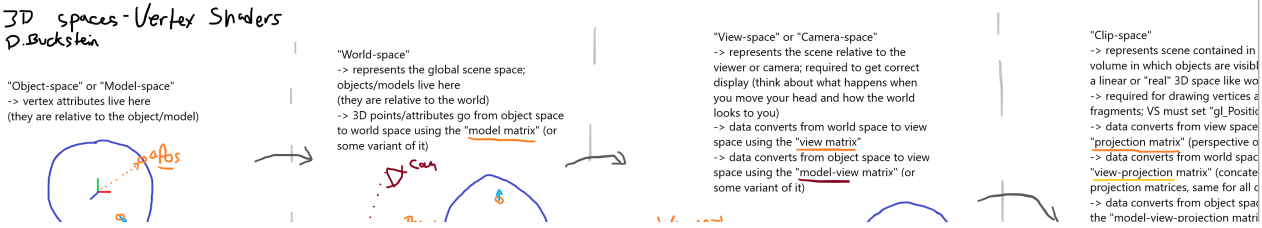
- **Simple overview of 3D models, vertices, primitive generation, rasterization, fragments and pixels**. The vertices define some 3D model or mesh, such as a plane or a sphere, which is broken down into primitives, such as triangles. The rasterizer produces fragments by "slicing" these shapes into tiny pieces (hence, "fragments"). Fragments are converted to pixels when they are written to the active framebuffer's target image. Note: there cannot be pixels without fragments, and there cannot be fragments without vertices! Also note: ***pixel != fragment***.

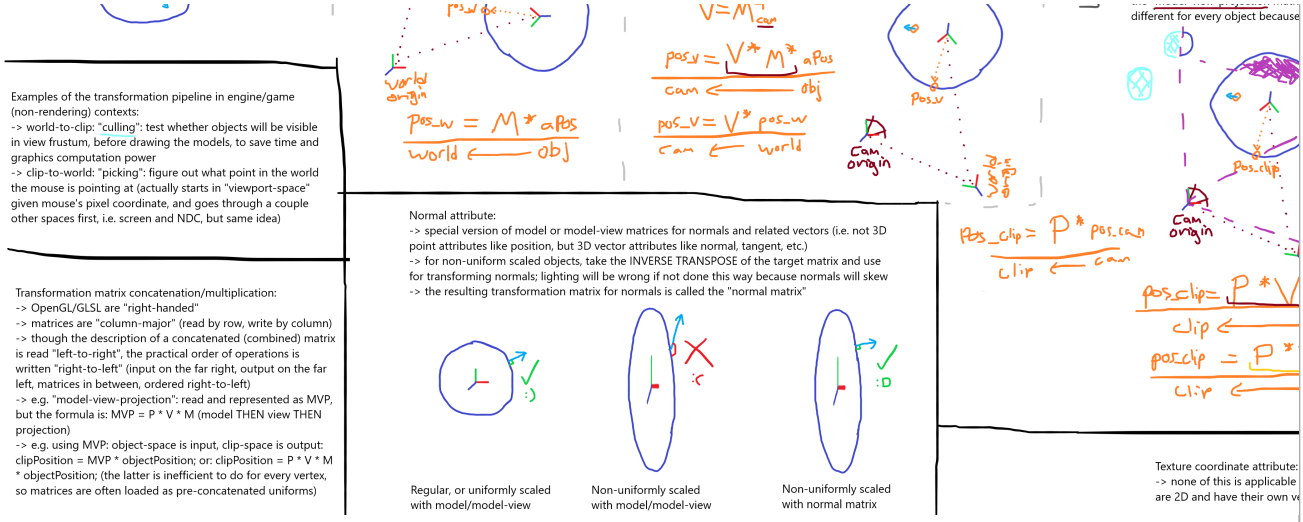






- Link to the full-sized, completed spaces diagram from class here.





- Link to the completed normal matrix explanation here:

More Transforms
D. Buckstein

1) "Normal matrix"
Understanding its construction and purpose

When transforming the POSITION attribute in any wa
entire matrix, as demonstrated previously. The NOR
however, is not a point in space; it is a vector, so we
to translate. Therefore, we can extract the upper-left
We can call this 'M' for "some matrix" but in this con

While certain linear transformation matrices (let's call them 'T'), such as the model, model-view and even their inverses, are provided to us by the engine, we must understand their mathematical construction:

$$T = \begin{bmatrix} (RS)_{3\times3} & \vec{t}_{3\times1} \\ \vec{0}^{T}_{1\times3} & 1 \end{bmatrix}$$

T: a transformation matrix is a 4x4 matrix
RS: the upper-left 3x3 portion is the product of a rotation (R) matrix and a scale (S) matrix
t: the upper-right 3x1 column-vector is a translation vector (position or offset relative to parent space)
0: the bottom-left 1x3 row-vector is just ZEROs
1: the bottom-right single element is just ONE (allows the translation component to take effect when concatenating/combining transformations)

The "normal matrix" is used to transform the normal such that it remains perpendicular to the surface. It is calculated as the INVERSE TRANSPOSE (or transpose inverse) of the selected 'M' matrix (again, either model or model-view).

If M does not have scale (i.e. just rotation), the inverse transpose cancels itself out and the matrix is unchanged:

$$M = R$$

$$(M^{-1})^{T} = (R^{-1})^{T} = (R^{T})^{T}$$

$$= (R^{-1})^{-1} = R = M$$

Therefore, applying the matrix to the normal only rotates it!

explitly define it as either the model (object-space to
or model-view (object-space to camera-space) matri

$$M = (RS)_{3\times3}$$

R (rotation) and S
3x3 matrices, mult
'M', the basis of tra
for the NORMAL a

A rotation matrix has two special properties:
1) its determinant is 1 (which is assumed, but you can test to confirm whether your 'M' represents just a rotation or a combined rotation x scale)
2) its inverse is very easy to calculate: it is equal to the matrix's transpose (superscript T)

A scale matrix also has two special properties:
1) since it is a diagonal matrix, its transpose is equal to the matrix itself
2) its inverse is easy to calculate: take the reciprocal of each element

$$S = \begin{bmatrix} s_x & & \\ & s_y & \\ & & s_z \end{bmatrix} \qquad S^{-1} = \begin{bmatrix}$$

If M is comprised of both rotation and scale, watch what h
when we calculate its inverse transpose:

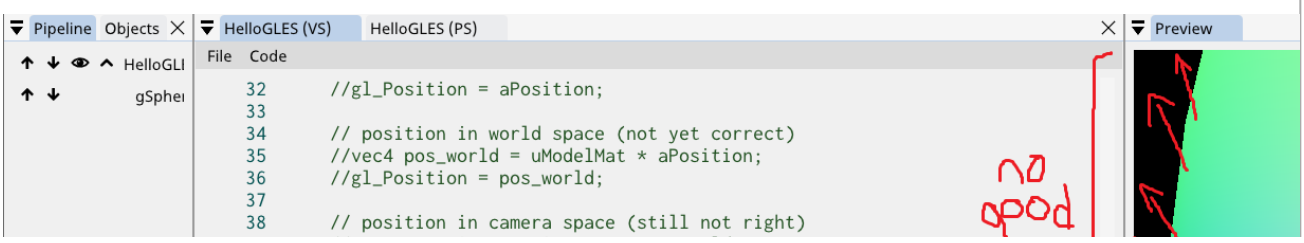
$$M = RS$$

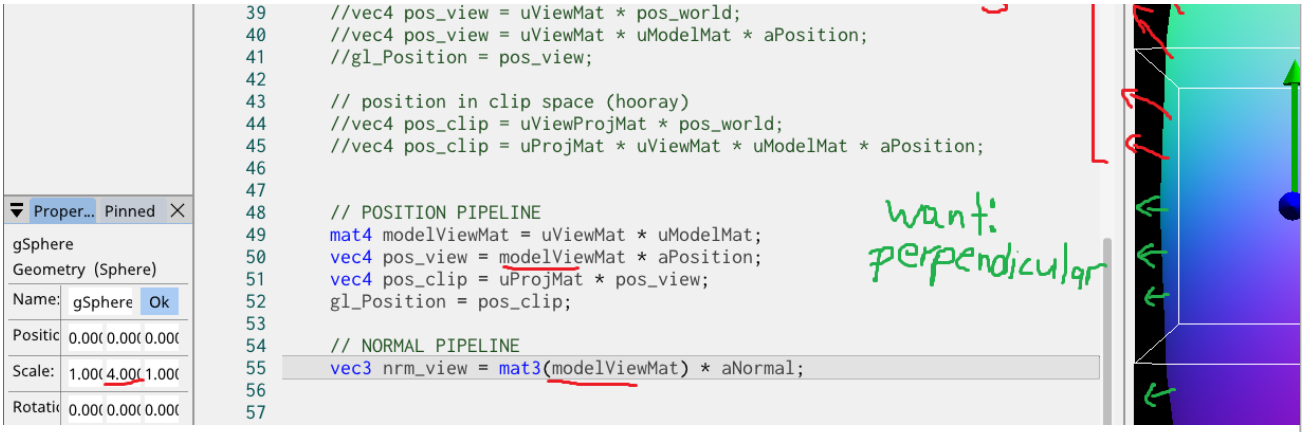
$$(M^{-1})^{T} = ((RS)^{-1})^{T} = (S^{-1}R$$

$$= (R^{-1})^{T}(S^{-1})^{T} = RS^{-1}$$

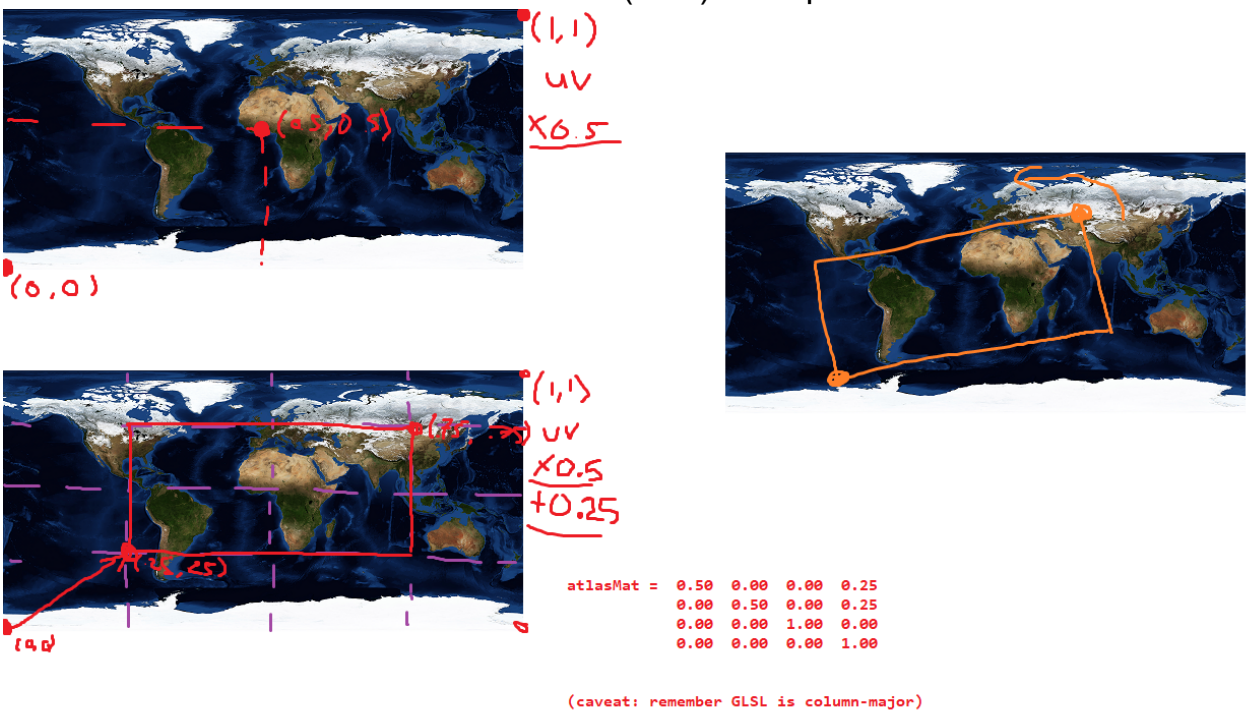
This means the scale part gets inverted, while the rotation
remains unchanged! The inverse scale has the effect of n
vector perpendicular to the surface:

- Visualization in SHADERed. The moral of the story is, when the geometry scales up, we want the normals to scale down:





- Atlas transform and texture coordinates (UVs) example:



7. ***Optimization***: Avoid expensive code wherever possible. In this case, you may rewrite functions where appropriate as there is no concept of some 'common' tab like in Shadertoy; the shaders (vertex and fragment) operate independently from each other. That said, still try to write and reuse flexible code wherever possible. In your demo video, showcase the differences between the four modes and discuss any opportunities for optimization that you can identify.

Bonus:
You are encouraged to complete one or more of the following bonus opportunities (rewards listed):

- ***Non-photorealistic shading (+1)***: Implement a non-photorealistic shading algorithm using texture ramps.
- ***Projection matrices (+1)***: Implement and demonstrate your own perspective and orthographic projection matrices, instead of using the built-in ones.
- ***Vertex deformation (+1)***: Implement a deformation effect within the vertex shader that

morphs vertices with respect to time.
- ***Shading language trifecta (+2)***: Implement the assignment in GLSL, GLSL ES and HLSL. The differences in the GLSL shaders should be minimal, if any, while the port to HLSL will require some familiarization with the new language and research; regardless, the logic and output of your shader effects should not change.

Coding Standards:
You are required to mind the following standards (penalties listed):

- ***Reminder: You may be referencing others' ideas and borrowing their code. Credit and provide a link to source materials (books, websites, forums, etc.) in your work wherever code from there is borrowed, and credit your instructor for the starter framework, even if it is adapted, modified or reworked (failure to cite sources and claiming source materials as one's own results in an instant final grade of F in the course)***. Recall that borrowed material, even when cited, is not your own and will not be counted for grades; therefore you must ensure that your assignment includes some of your own contributions and substantial modification from what is provided in the book. ***This principle applies to all evaluations***.
- ***Reminder: You must use version control consistently (zero on organization)***. Even though you are using a platform which allows you to save your work online, you must frequently commit and back up your work using version control. In your repository, create a new text file for each tab used in your selected tool (e.g. "vertex.glsl" and "fragment.glsl") and copy your code there. This will also help you track your progress developing your shaders. Remember to work on a new branch for each assignment.
- ***The assignment should be completed using SHADERed (https://shadered.org/) or some other interface that allows easy editing of both GLSL vertex and fragment shaders (zero on assignment)***. You must use a platform that gives you the easy ability to edit both ***vertex and fragment shaders*** with real-time feedback. The modern language we are using is ***GLSL*** and you must be able to edit shaders using either GLSL 4.50 (standard) or GLSL ES 3.00 (WebGL 2.0). Other viable interfaces that can include KickJS and ShaderFrog; if you can find another one that satisfies this requirement, please propose it to your instructor before using (also good luck).
- ***Use the most efficient methods (-1 per inefficient/unnecessary practices)***: Your goal is to implement optimized and thoughtful shader code instead of just implementing the demo as-is. Use inefficient functions and methods sparingly and out of necessity (including but not limited to conditionals, square roots, etc.). Put some thought into everything you do and figure out if there are more efficient ways to do it.

- ***Every section, block and line of code must be commented (-1 per ambiguous section/block/line)***. Clearly state the intent, the 'why' behind each section, block and even line (or every few related lines) of code. This is to demonstrate that you can relate what you are doing to the subject matter.
- ***Add author information to the top of each code file (-1 for each omission)***. If you

have a license, include the boiler plate template (fill it in with your own info) and add a separate block with: 1) the name and purpose of the file; and 2) a list of contributors and what they did.

Points 5

Submitting a text entry box or a website url

Due	For	Available from	Until
-	Everyone	-	-

GraphicsAnimation-Master-Range

Criteria	Ratings			Pts
IMPLEMENTATION: Architecture & Design Practical knowledge of C/C++/API/framework programming, engineering and architecture within the provided framework or engine.	**1 to >0.5 pts** **Full points** Strong evidence of efficient and functional C/C++/API/framework code implemented for this assignment; architecture, design and structure are largely both efficient and functional.	**0.5 to >0.0 pts** **Half points** Mild evidence of efficient and functional C/C++/API/framework code implemented for this assignment; architecture, design and structure are largely either efficient or functional.	**0 pts** **Zero points** Weak evidence of efficient and functional C/C++/API/framework code implemented for this assignment; architecture, design and structure are largely neither efficient nor functional.	1 pts
IMPLEMENTATION: Content & Material Practical knowledge of content relevant to the discipline and course (e.g. shaders and effects for graphics, animation algorithms and techniques, etc.).	**1 to >0.5 pts** **Full points** Strong evidence of efficient and functional course- and discipline-specific algorithms and techniques implemented for this assignment; discipline-relevant algorithms and techniques are largely both efficient and functional.	**0.5 to >0.0 pts** **Half points** Mild evidence of efficient and functional course- and discipline-specific algorithms and techniques implemented for this assignment; discipline-relevant algorithms and techniques are largely either efficient or functional.	**0 pts** **Zero points** Weak evidence of efficient and functional course- and discipline-specific algorithms and techniques implemented for this assignment; discipline-relevant algorithms and techniques are largely neither efficient nor functional.	1 pts
DEMONSTRATION: Presentation & Walkthrough Live presentation and walkthrough of code, implementation, contributions, etc.	**1 to >0.5 pts** **Full points** Strong evidence of accuracy and confidence in a live walkthrough of code discussing requirements and high-level contributions; walkthrough is largely both accurate and confident.	**0.5 to >0.0 pts** **Half points** Mild evidence of accuracy and confidence in a live walkthrough of code discussing requirements and high-level contributions; walkthrough is largely either accurate or confident.	**0 pts** **Zero points** Weak evidence of accuracy and confidence in a live walkthrough of code discussing requirements and high-level contributions; walkthrough is largely neither accurate nor confident.	1 pts
DEMONSTRATION: Product & Output Live showing and explanation of final working implementation, product and/or outputs.	**1 to >0.5 pts** **Full points** Strong evidence of correct and stable final product that runs as expected; end result is largely both correct and stable.	**0.5 to >0.0 pts** **Half points** Mild evidence of correct and stable final product that runs as expected; end result is largely either correct or stable.	**0 pts** **Zero points** Weak evidence of correct and stable final product that runs as expected; end result is largely neither correct nor stable.	1 pts

Criteria	Ratings			Pts
ORGANIZATION: Documentation & Management Overall developer communication practices, such as thorough documentation and use of version control.	**1 to >0.5 pts** **Full points** Strong evidence of thorough code documentation and commenting, and consistent organization and management with version control; project is largely both documented and organized.	**0.5 to >0.0 pts** **Half points** Mild evidence of thorough code documentation and commenting, and consistent organization and management with version control; project is largely either documented or organized.	**0 pts** **Zero points** Weak evidence of thorough code documentation and commenting, and consistent organization and management with version control; project is largely neither documented nor organized.	1 pts
BONUSES Bonus points may be awarded for extra credit contributions.	**0 pts** **Points awarded** If score is positive, points were awarded for extra credit contributions (see comments).		**0 pts** **Zero points**	0 pts
PENALTIES Penalty points may be deducted for coding standard violations.	**0 pts** **Points deducted** If score is negative, points were deducted for coding standard violations (see comments).		**0 pts** **Zero points**	0 pts
				Total Points: 5