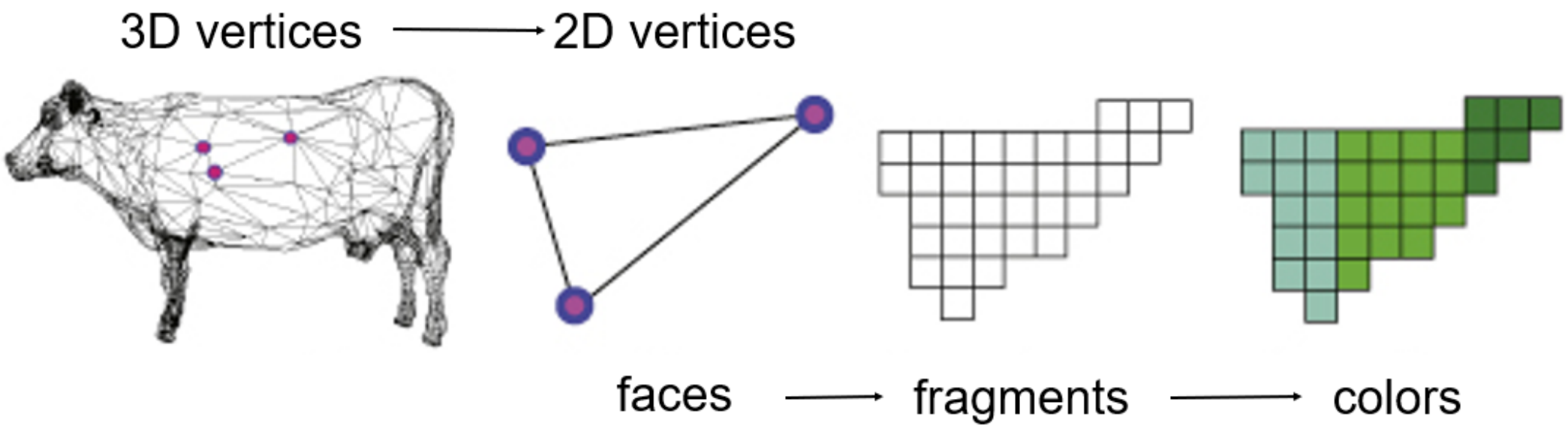
UNIVERSITY OF MINNESOTA
Driven to Discover®

Graphics Pipeline and Shaders

CSCI 4611: Programming Interactive Computer Graphics and Games

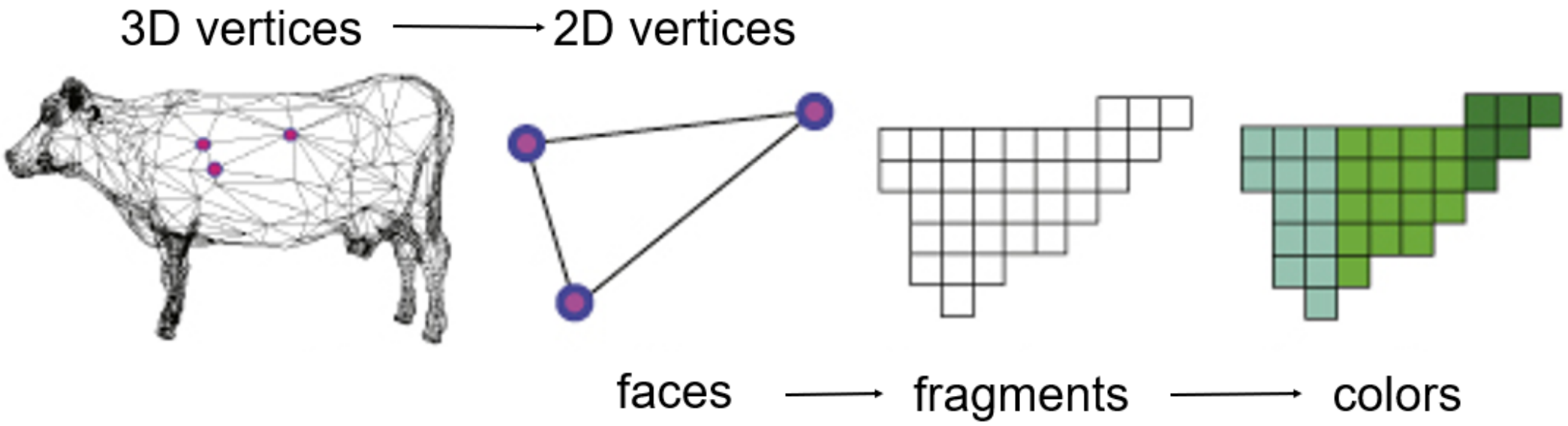
Evan Suma Rosenberg | CSCI 4611 | Fall 2022

Graphics Pipeline (Simplified)



Graphics Pipeline (Simplified)

vertex shader



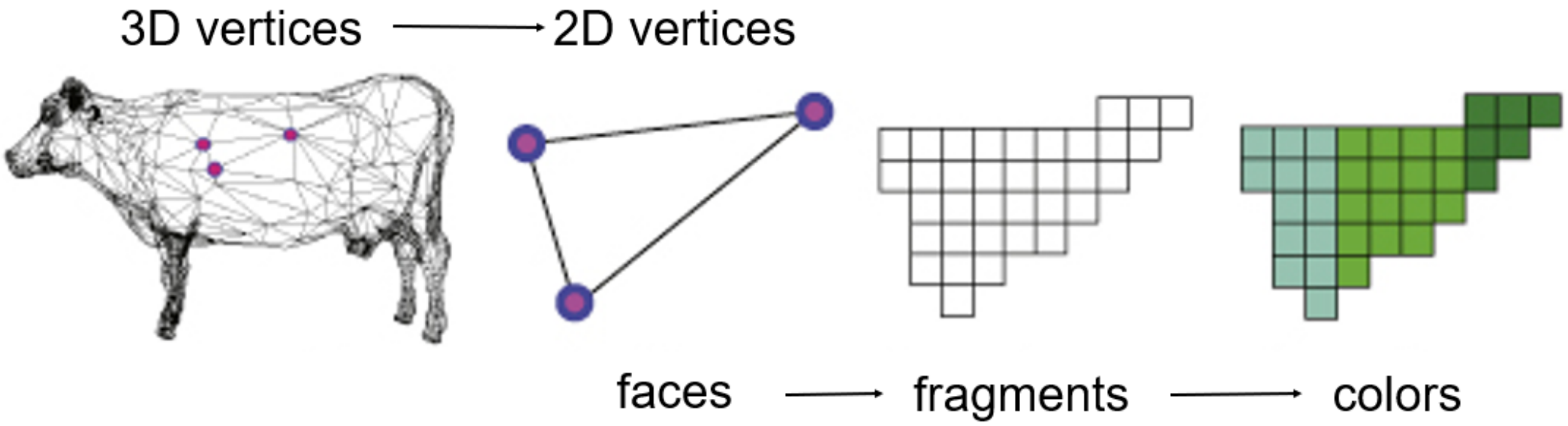
rasterizer

fragment shader

Graphics Pipeline (Simplified)

Customizable via shader programs!

vertex shader



rasterizer

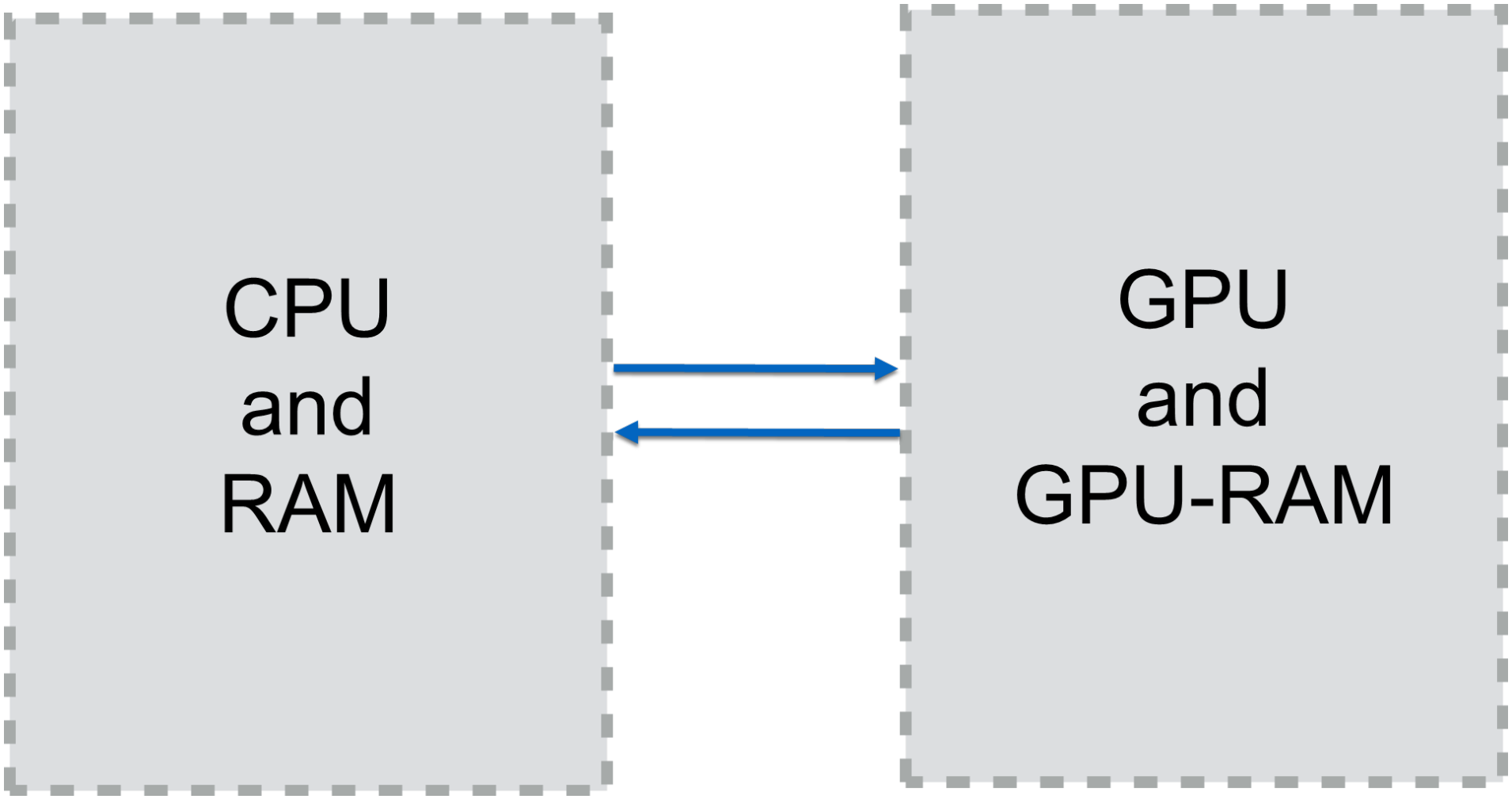
fragment shader

Viewed in Terms of Hardware and Software

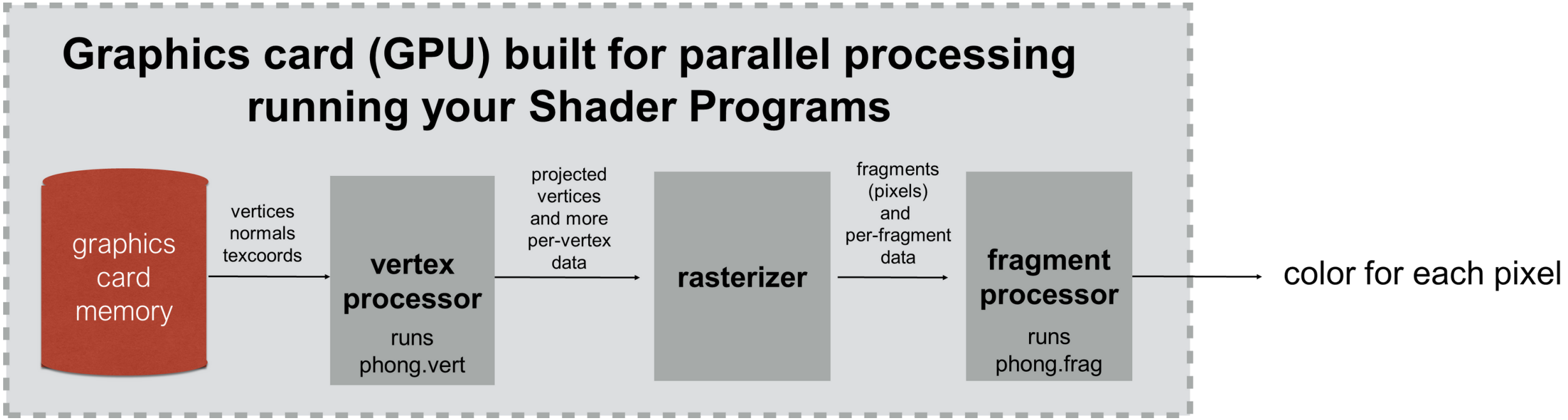
CPU running your app

```
initialize(): void
{
    // Compile and link a new shader program (a vertex shader + fragment shader)
    import phongVertexShader from './shaders/phong.vert'
    import phongFragmentShader from './shaders/phong.frag'
    const shader = new gfx.ShaderProgram(phongVertexShader, phongFragmentShader);
    shader.initialize(this.gl);
}

draw(): void
{
    // Makes the pre-compiled shader program active on the GPU
    gl.useProgram(shader.getProgram());

    // Set any "uniform" data for the shader, like the model, view, and projection matrices, light position(s),
etc..

    // Set array attributes such as the mesh's vertices, normals, and texcoords for input

    // Instruct the shader to draw elements (triangles)
}
```

Viewed in Terms of Hardware and Software



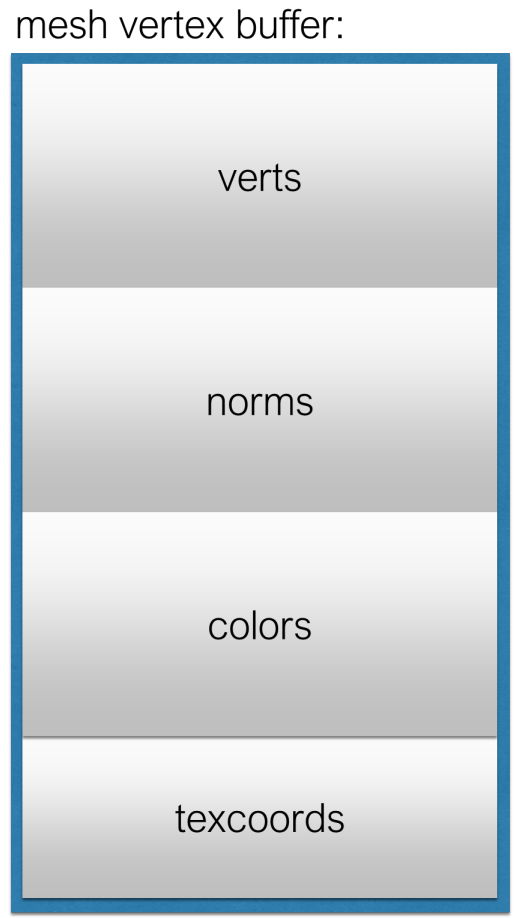
Key Stages of the Pipeline for Shader Programs



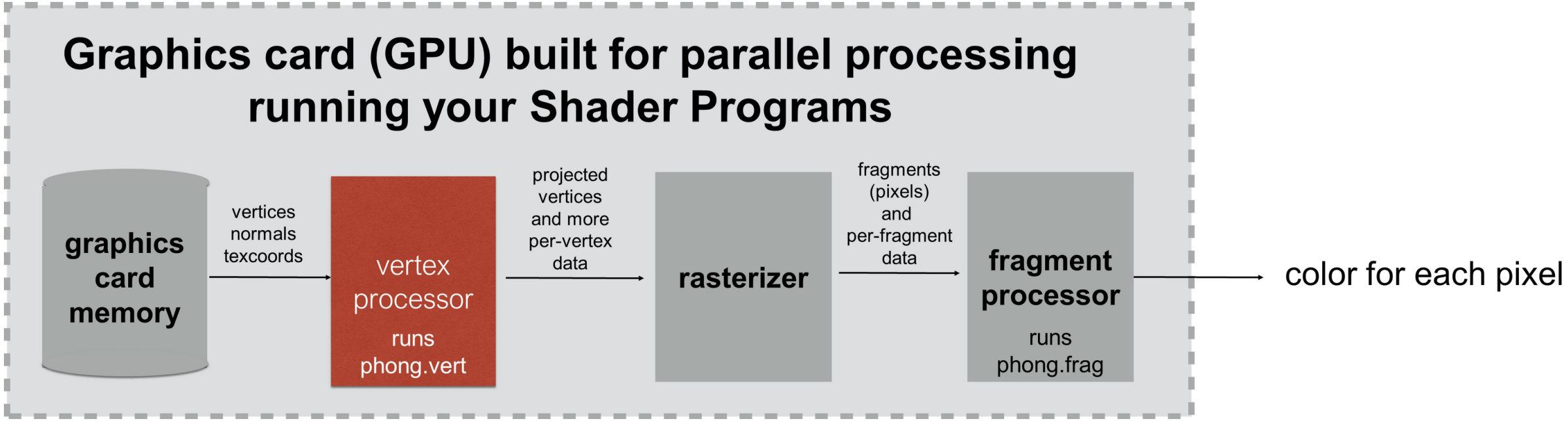
Inside a Mesh Class

```
setVertices(vertices: number[]): void
{
        this.gl.bindBuffer(this.gl.ARRAY_BUFFER, this.positionBuffer);

        this.gl.bufferData(this.gl.ARRAY_BUFFER, new
Float32Array(vertices),
            this.gl.DYNAMIC_DRAW
        );
}
```

GPU memory:



Key Stages of the Pipeline for Shader Programs



- We get to program this stage! The small program we write is called a **vertex shader**.
- It will run in parallel on the GPU once for each vertex.

  The input will be the vertex position, normal, texcoords, and possibly other data stored in GPU memory.

  The output is (typically) a projected version of the vertex.

A Typical Vertex Shader Program

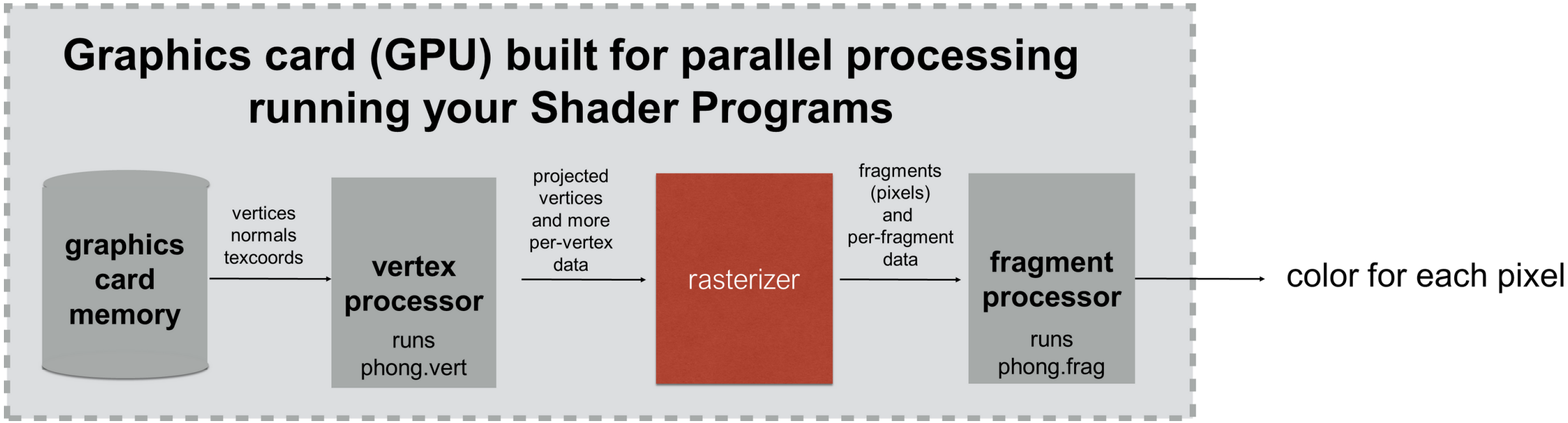
- We will cover syntax, handling normals, and more soon.
- For now, let's just look at the highlighted lines, which show a typical program for rendering 3D geometry.

input: vertex position
(from the raw data in the mesh)

output: position multiplied by proj, model, & view matrices.

```
1  #version 330
2
3  uniform mat4 modelViewMatrix;
4  uniform mat4 normalMatrix;
5  uniform mat4 projectionMatrix;
6
7  layout(location = 0) in vec3 vertex;
8  layout(location = 1) in vec3 normal;
9
10 out vec3 Position;
11 out vec3 Normal;
12
13 void main() {
14     Position = (modelViewMatrix * vec4(vertex,1)).xyz;
15     Normal = normalize((normalMatrix * vec4(normal,0)).xyz);
16     gl_Position = projectionMatrix * modelViewMatrix * vec4(vertex,1);
17 }
```

Key Stages of the Pipeline for Shader Programs



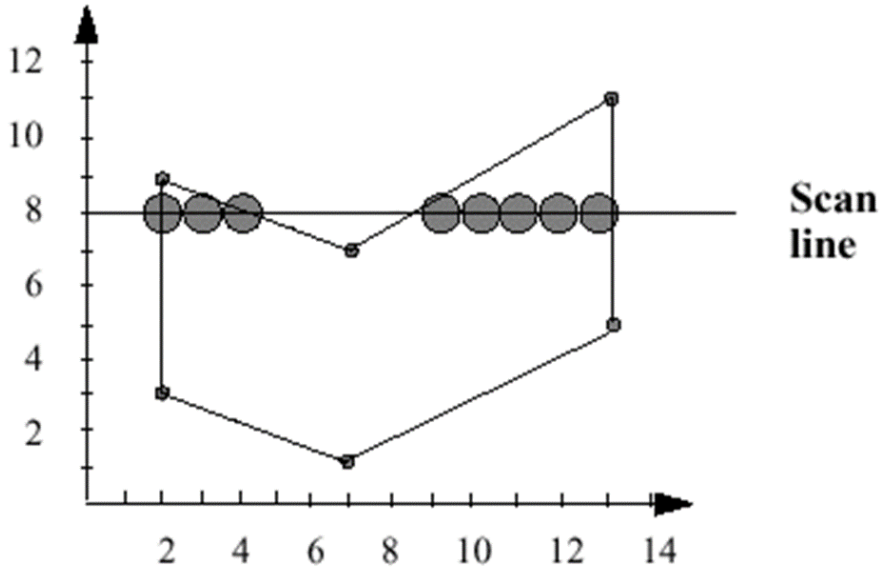
What does the rasterizer do?

Fragment creation:

Find the bounding box
(min/max *x*, min/max *y*)

For each scan line in the bounding box,
find intersections with polygon edges

Create fragments for all pixels between intersections

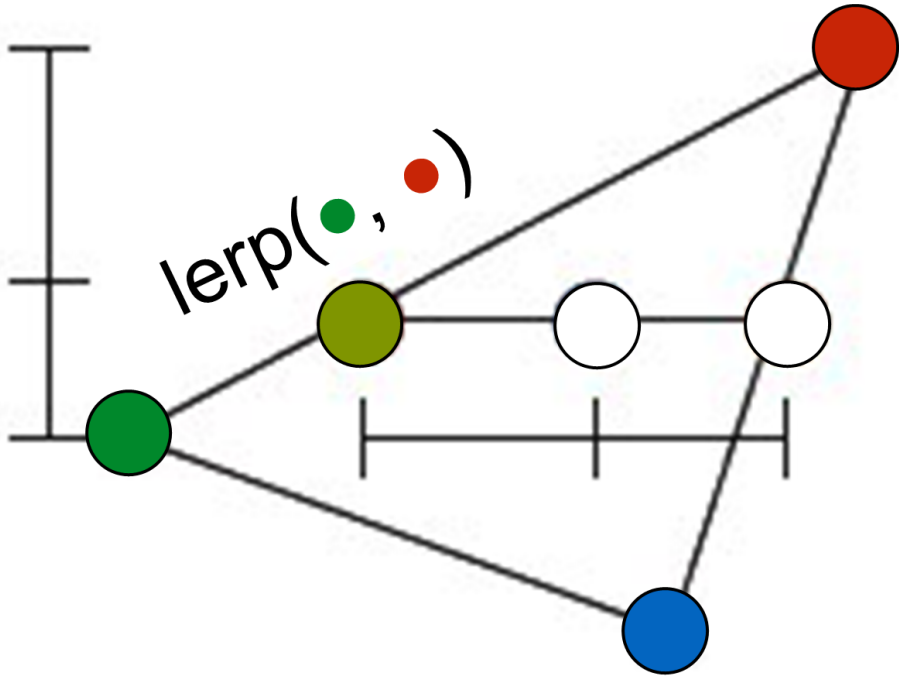


What does the rasterizer do?

Interpolation:

Use y distance to interpolate the two end points of a scan line.

Then use *x* distance to interpolate interior colors, texture coordinates, normals and/or other per-vertex data.

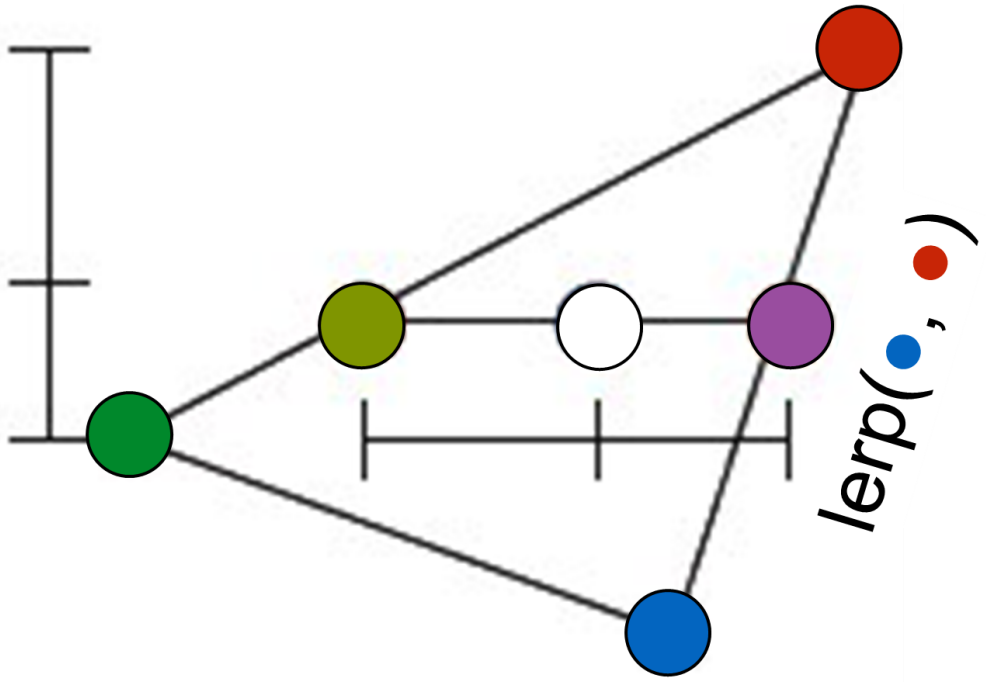


What does the rasterizer do?

Interpolation:

Use y distance to interpolate the two end points of a scan line.

Then use *x* distance to interpolate interior colors, texture coordinates, normals and/or other per-vertex data.

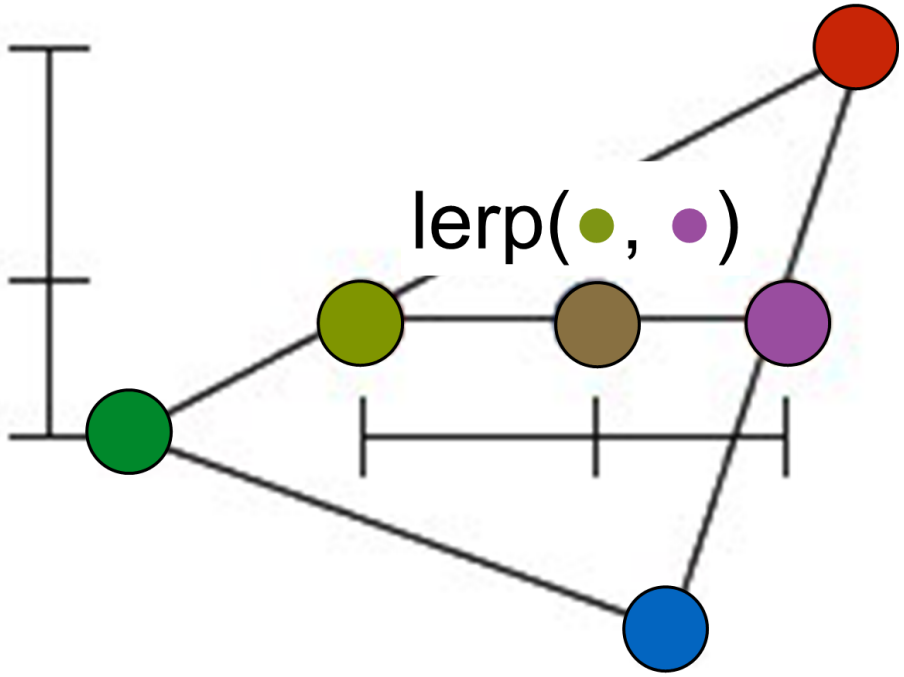


What does the rasterizer do?

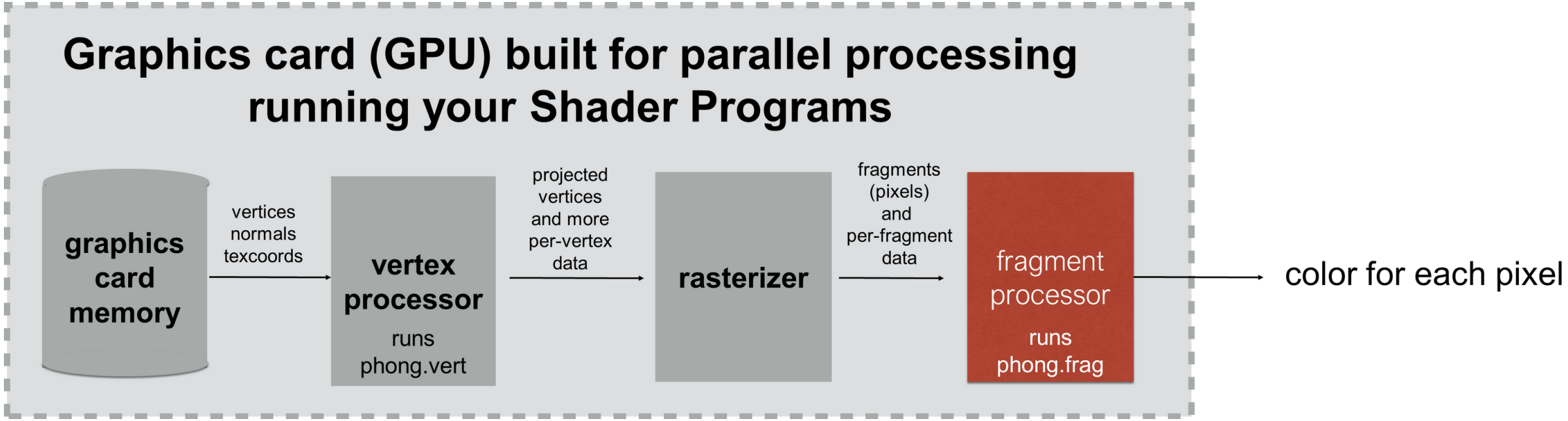
Interpolation:

Use y distance to interpolate the two end points of a scan line.

Then use *x* distance to interpolate interior colors, texture coordinates, normals and/or other per-vertex data.



Key Stages of the Pipeline for Shader Programs



- We get to program this stage! The small program we write is called a **fragment shader**.
- It will run in parallel on the GPU once for each fragment (pixel).
- The input to the fragment shader is determined by the vertex shader we wrote earlier. We can send variables (defined per-vertex) from the vertex shader to the fragment shader.

  Input variables get automatically **interpolated** by the rasterizer before they are read into the fragment shader.

  Fragment shaders are the last step in the pipeline, and there is just a single output — the color of the pixel!

The Simplest Possible Fragment Shader

We will cover syntax and passing data from the vertex shader and from the app to the fragment shader soon.

For now, let's just look at the highlighted lines, which output a black color for the pixel.

```
1  #version 330
2
3  in vec3 Position;
4  in vec3 Normal;
5
6  out vec4 color;
7
8  uniform vec3 lightPosition;
9  uniform vec4 Ia, Id, Is;
10
11 uniform vec4 ka, kd, ks;
12 uniform float s;
13
14
15 void main() {
16     color = vec4(0,0,0,1);
17 }
```

GLSL

Vertex and fragment shaders are written in the OpenGL Shading Language (GLSL).

The syntax resembles C/C++.

GLSL has evolved with different versions of the API. WebGL uses version 3.0 ES.

https://www.khronos.org/files/opengles3-quick-reference-card.pdf

The first line of the shader must declare the GLSL version number.

```
#version 300 es
```

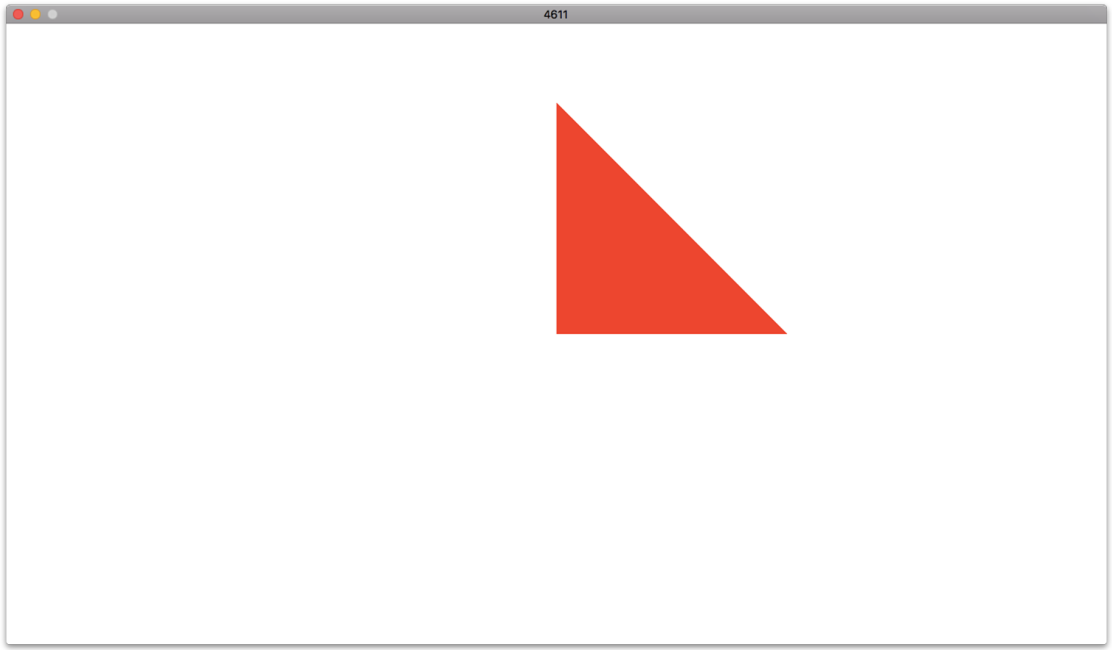
Hello world!

Vertex shader:

```
#version 330
uniform mat4 modelView;
uniform mat4 projection;
in vec3 vertex;
void main() {
  gl_Position = projection
    * modelView * vec4(vertex,1);
}
```

Fragment shader:

```
#version 330
out vec4 color;
void main() {
  color = vec4(1,0,0, 1);
}
```



Hello world!

Vertex shader:

```
#version 330
uniform mat4 modelView;
uniform mat4 projection;
in vec3 vertex;
void main() {
  gl_Position = projection
    * modelView * vec4(vertex,1);
}
```

Fragment shader:

```
#version 330
out vec4 color;
void main() {
  color = vec4(1,0,0, 1);
}
```

- All shader programs have type void main().
- Input and output happens on global variables labeled as in/out.
- Vertex shader must write to vec4 gl_Position.
- Fragment shader must have an out vec4.

Programming in GLSL

Data types (incomplete list)

- Scalars:
  float, int, uint, bool
- Vectors:
  vec2, vec3, vec4
- Matrices:
  mat2, mat3, mat4
- Textures:
  sampler2D, sampler3D

Array indexing:

- v[1], m[2][3], etc.
- Can also use v.x, v.y, v.z, v.w
  (or r,g,b,a, or s,t,p,q)
- Control flow same as C/C++:
  if/else, switch/case/default, for, while, do/while, break, continue

More Vectors in GLSL

- Other types:
  bvec2, ivec3, uvec4, …
- "Swizzling:"
  vec2 a;
  vec4 b = a.xyxx;
  vec3 c = b.zyx;
  vec4 d = a.xxxx + c.yxzy;
- Flexible construction:
  vec2 p;
  vec4 q = vec4(p, 0, 0);
  vec4 r = vec4(q.xyz, 1);

Functions

```
// usual C/C++ syntax
float square(float x) {
  return x*x;
}
// but you can also have output arguments
void multiOut(out float x, out float y) {
  x = 1;
  y = 2;
}
// or take an argument and change it
void doubleIt(inout float x) {
  x *= 2;
}
// No recursion allowed! (hard for GPU)
```

Built-in Functions

- All the usual math functions (trig, exponentials, etc.)
- Geometric functions:
  length, distance, dot, cross, normalize, reflect, refract
- Other handy functions:
  modf, clamp, mix, step/smoothStep, lessThan, etc.
  any/all/not, …

Textures in Shaders

Textures usually store an image, but really they could store any data, which makes them ***very*** useful as a way to pass all kinds of data into shaders!

Textures in Shaders

Most interesting shader effects make use of textures in the fragment shader.

Application side:

- The texture to use inside the shader is a parameter that in passed into the shader program, similar to the regular variables, except that textures use **bindTexture()**.

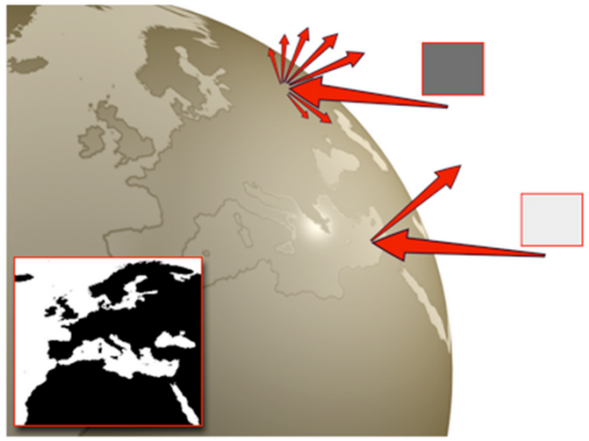
GLSL side:

- Calculate the 2D texture coordinate to use for the particular fragment.

  Look up the color in the texture image at those coordinates using the **texture()** function.

Things you can do with shaders

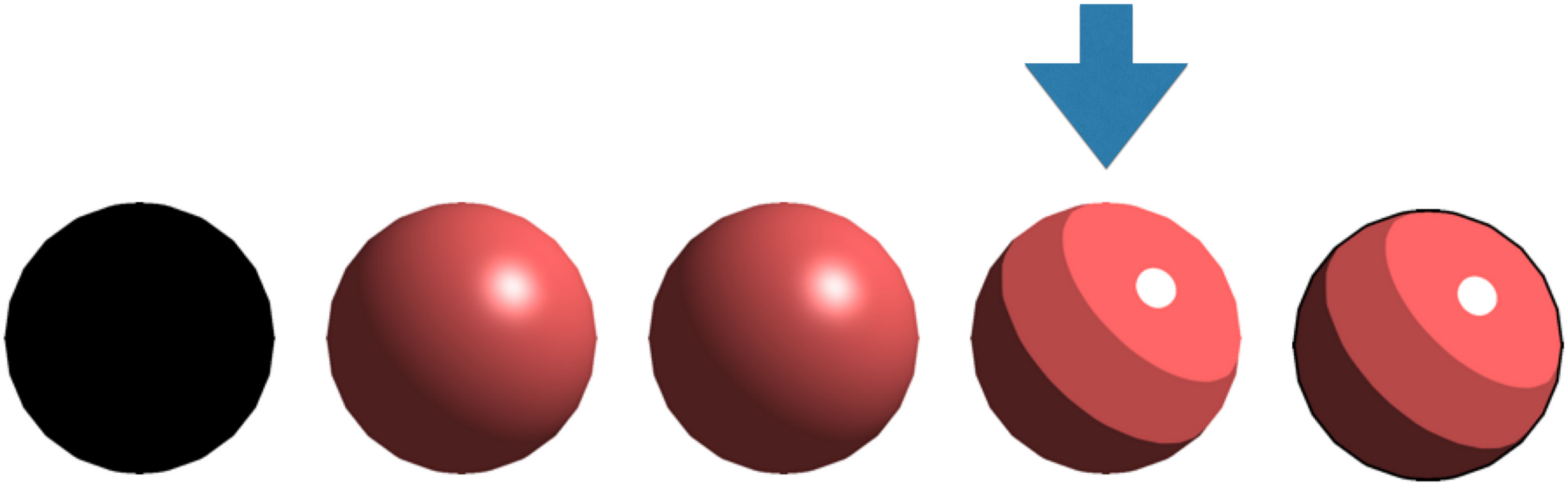
Artistic shading



Spatially varying shininess

Per-pixel normals

Texture “Ramps” in Assignment 5

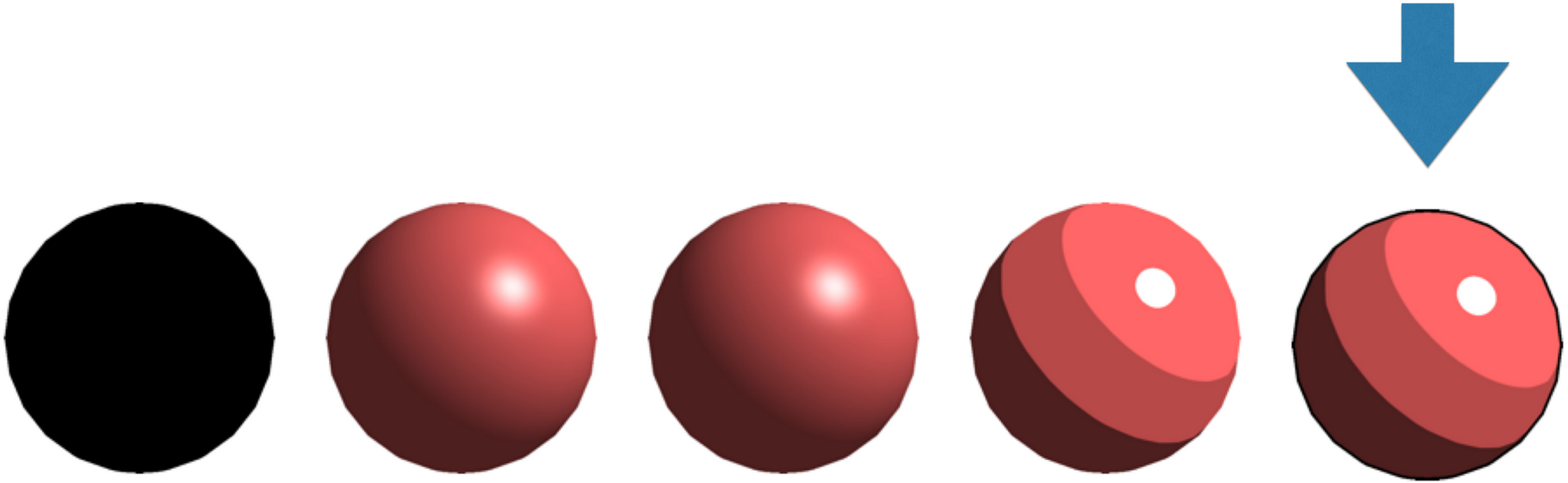


Texture “Ramps” in Assignment 5

standardDiffuse.png

toonDiffuse.png

Silhouette Shading in Assignment 5



Silhouette Outlines Using the Stencil Buffer

- Once the fragment shader has processed the fragment, a **stencil test** can be executed that has the option to discard fragments.
- The stencil test is based on the content of yet another buffer that we can update during rendering to achieve interesting effects.

Silhouette Outlines Using the Stencil Buffer

1. Initialize the stencil buffer to all zeros.

Silhouette Outlines Using the Stencil Buffer

1. Initialize the stencil buffer to all zeros.

2. Draw object using the toon shader.

3. Any pixels rendered by the toon fragment shader have their value in the stencil buffer set to one.

Silhouette Outlines Using the Stencil Buffer

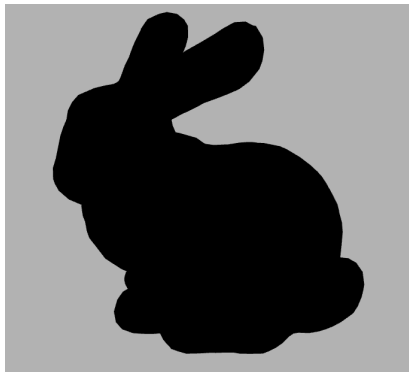
1. Initialize the stencil buffer to all zeros.

2. Draw object using the toon shader.

3. Any pixels rendered by the toon fragment shader have their value in the stencil buffer set to one.

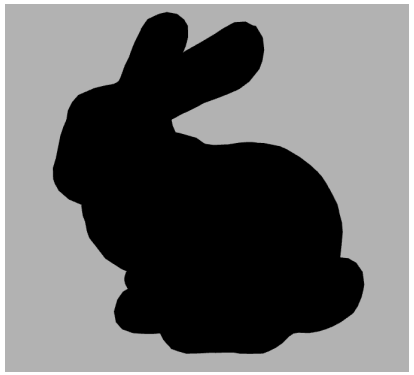
Silhouette Outlines Using the Stencil Buffer

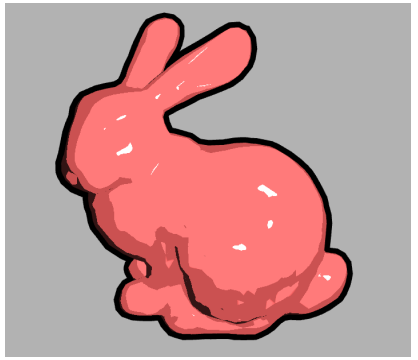
1. Initialize the stencil buffer to all zeros.
2. Draw object using the toon shader.
3. Any pixels rendered by the toon fragment shader have their value in the stencil buffer set to one.
4. Draw the object again using the outline shader.
5. The outline vertex shader enlarges the object by a specified thickness.



Silhouette Outlines Using the Stencil Buffer

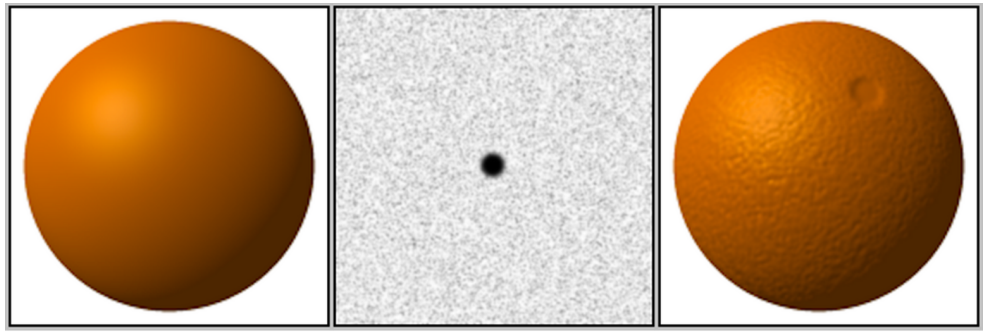
1. Initialize the stencil buffer to all zeros.
2. Draw object using the toon shader.
3. Any pixels rendered by the toon fragment shader have their value in the stencil buffer set to one.
4. Draw the object again using the outline shader.
5. The outline vertex shader enlarges the object by a specified thickness.
6. Any pixels with a non-zero value in the stencil buffer are discarded.
7. The outline fragment shader colors the remaining pixels as solid black.

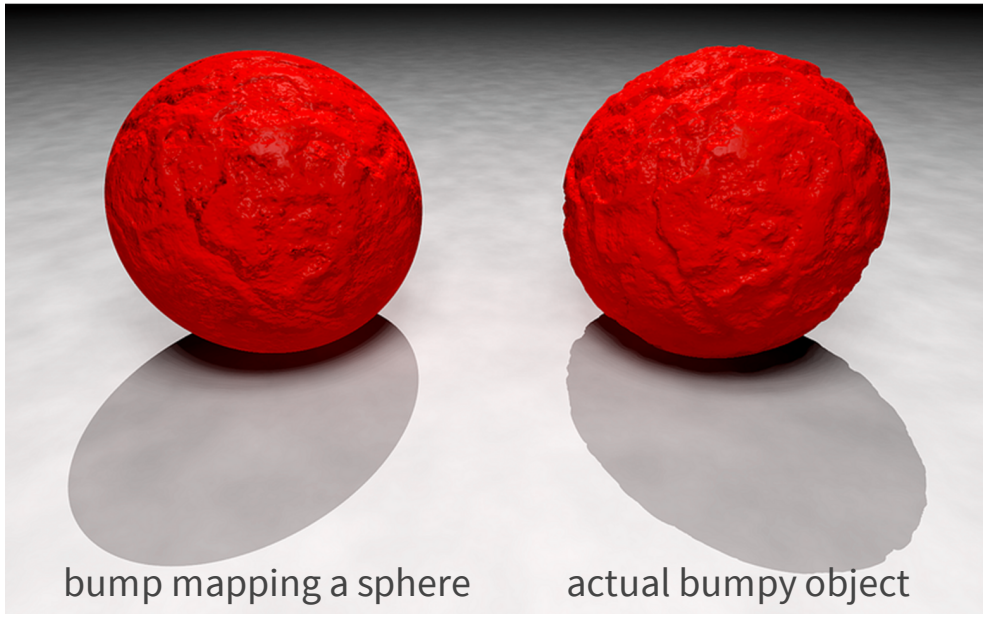




Bump Mapping

- Makes it possible to "fake" fine variations in the surface shape.
- Similar to texture mapping, the variation comes from detail stored in an image.
- Rather than storing colors, the image stores a height offset.
- You can use the height offset to modify the surface normal at each pixel inside a fragment shader.
- The quality is not as good as actually modeling the geometry of each bump, but it is much faster!





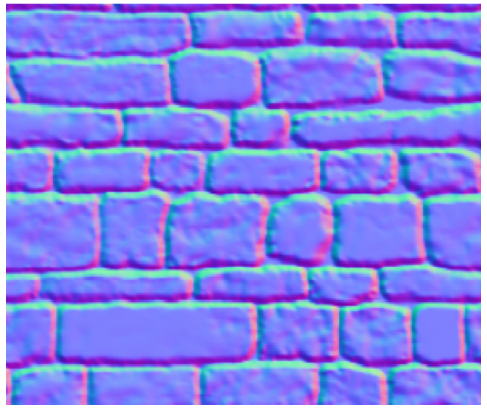
bump mapping a sphere actual bumpy object

Normal Mapping

- Similar to bump mapping
- Rather than storing a height, the texture passed into the fragment shader stores an actual normal.
- <x,y,z> are stored as <r,g,b>

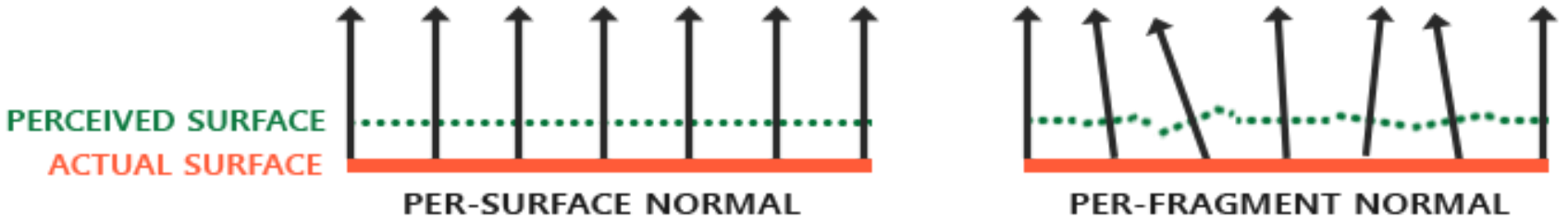
base texture

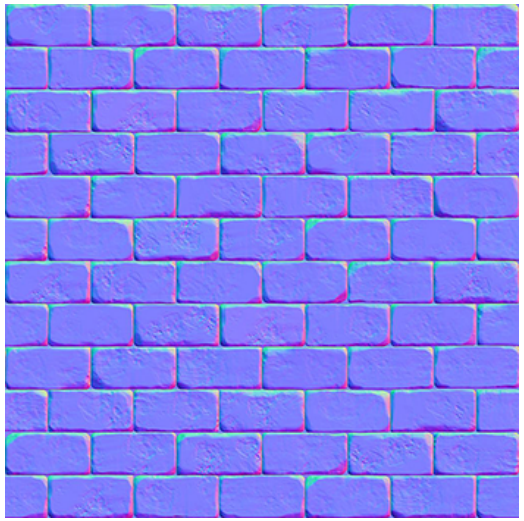
bump map

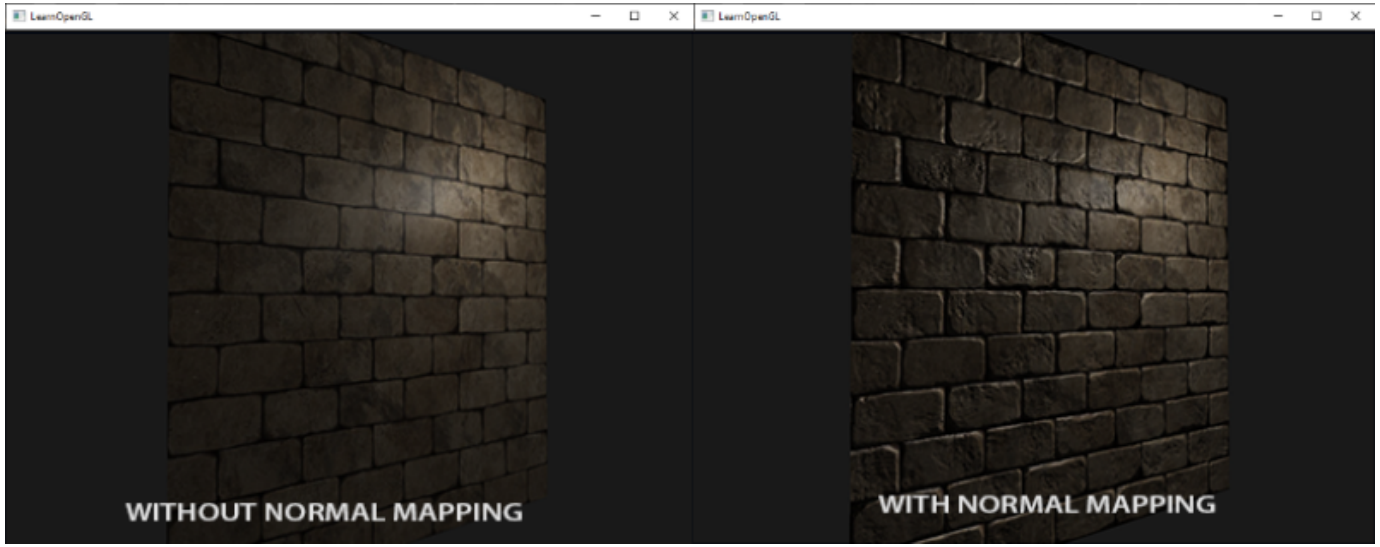


normal map

Normal Mapping

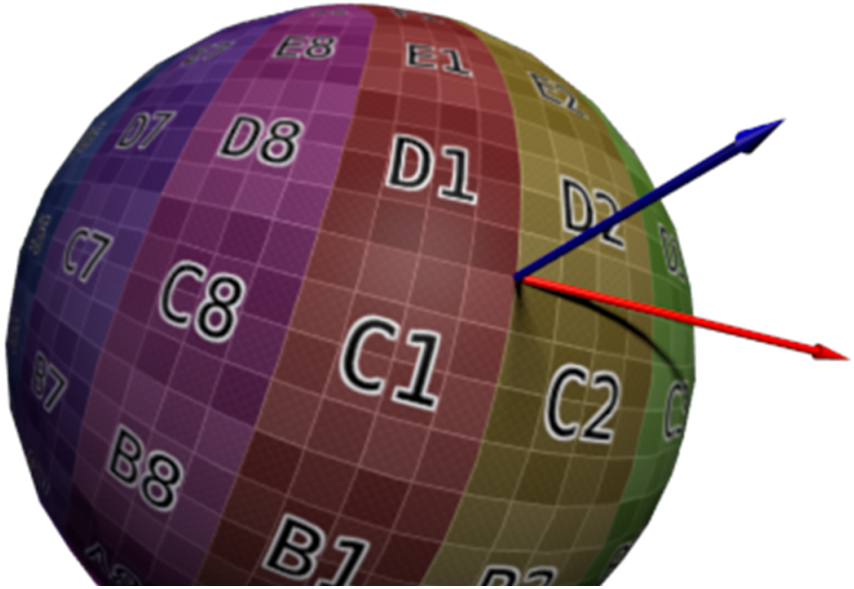






Bump/Normal Mapping in Practice

- To do bump mapping or normal mapping, you need to define not just a normal, but also a tangent vector.



- These tutorials have nice descriptions of how to calculate the tangent for any generic triangle mesh:

http://www.opengl-tutorial.org/intermediate-tutorials/tutorial-13-normal-mapping/

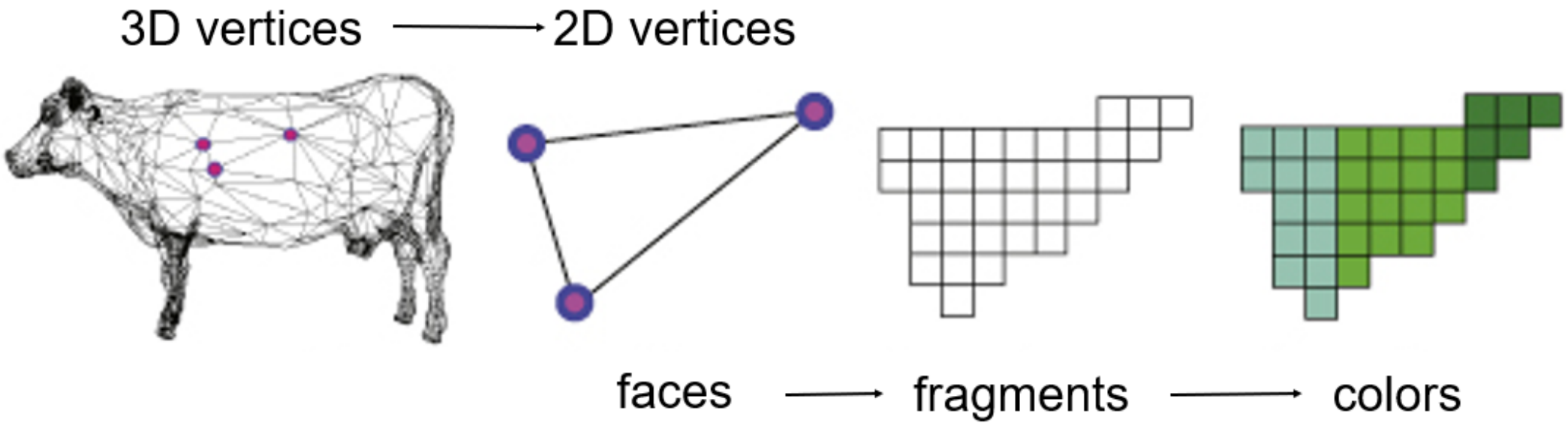
https://learnopengl.com/Advanced-Lighting/Normal-Mapping

Important Concept #1

Vertex and fragment shaders work as a pair.

Simplified Pipeline

vertex shader



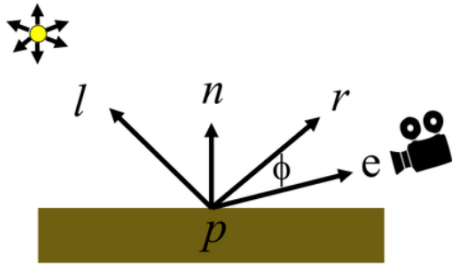
rasterizer

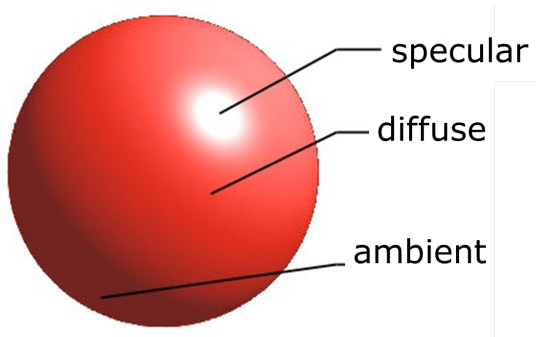
fragment shader

Important Concept #2

The Lighting Equation

The Lighting Equation





$$light\_intensity = \underbrace{k_a\ I_a}_{\text{ambient}} + \underbrace{k_d\ I_d\ (n \cdot l)}_{\text{diffuse}} + \underbrace{k_s\ I_s\ (h \cdot n)^s}_{\text{specular}}$$

or

$$(e \cdot r)^s$$

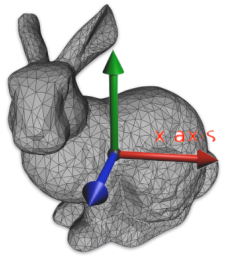
= because we can have colored lights or objects, these are 3D arrays (vec3) of floats for r, g, b

If there are N lights, then the **total illumination** for a point is the sum of the equation above for all lights.

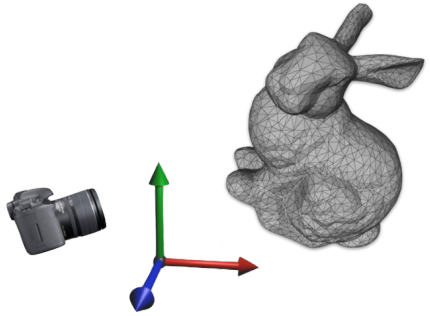
Important Concept #3

The five important matrices used in shader programs.

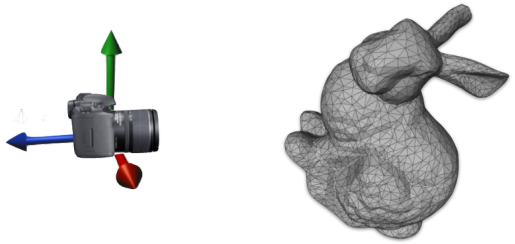
Coordinate Spaces



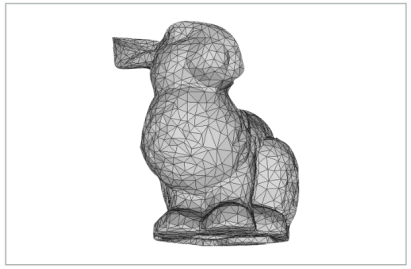
Object space



World space

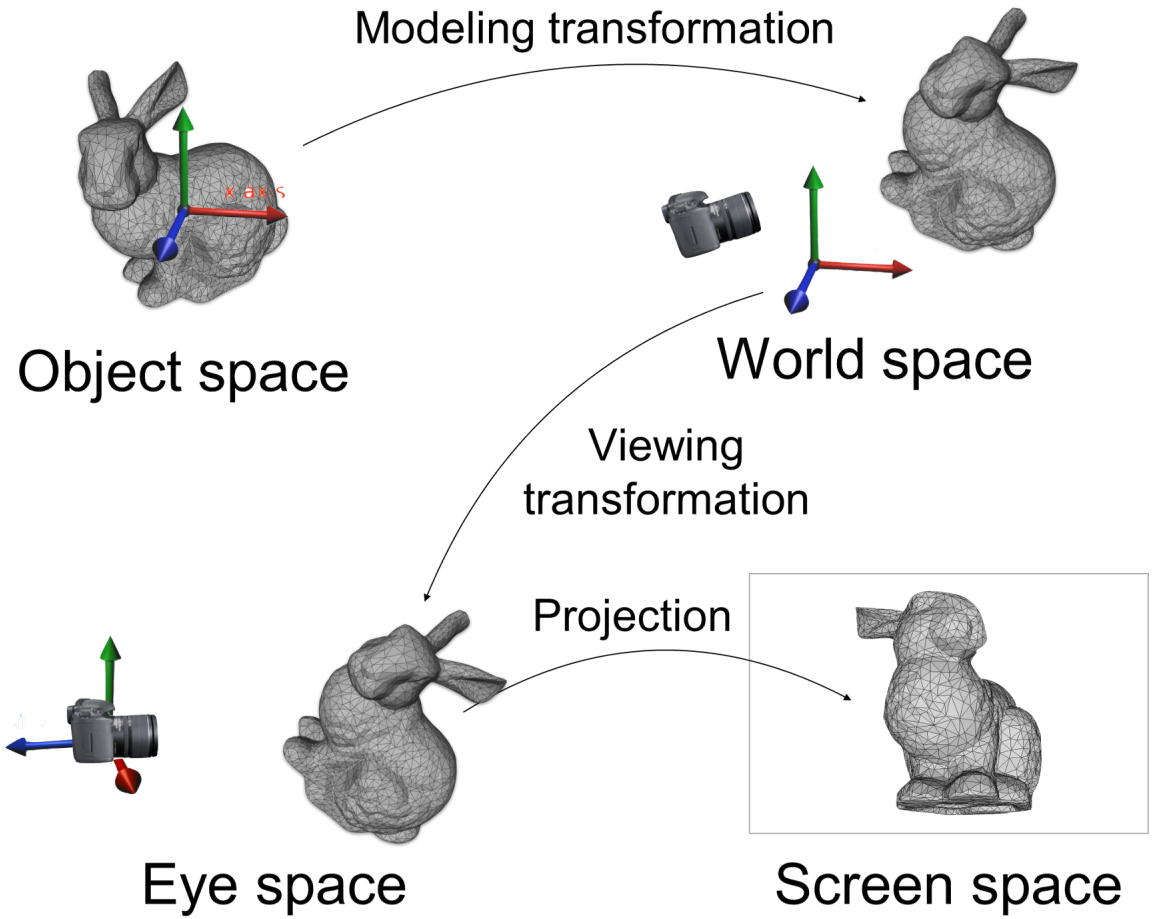


Eye space



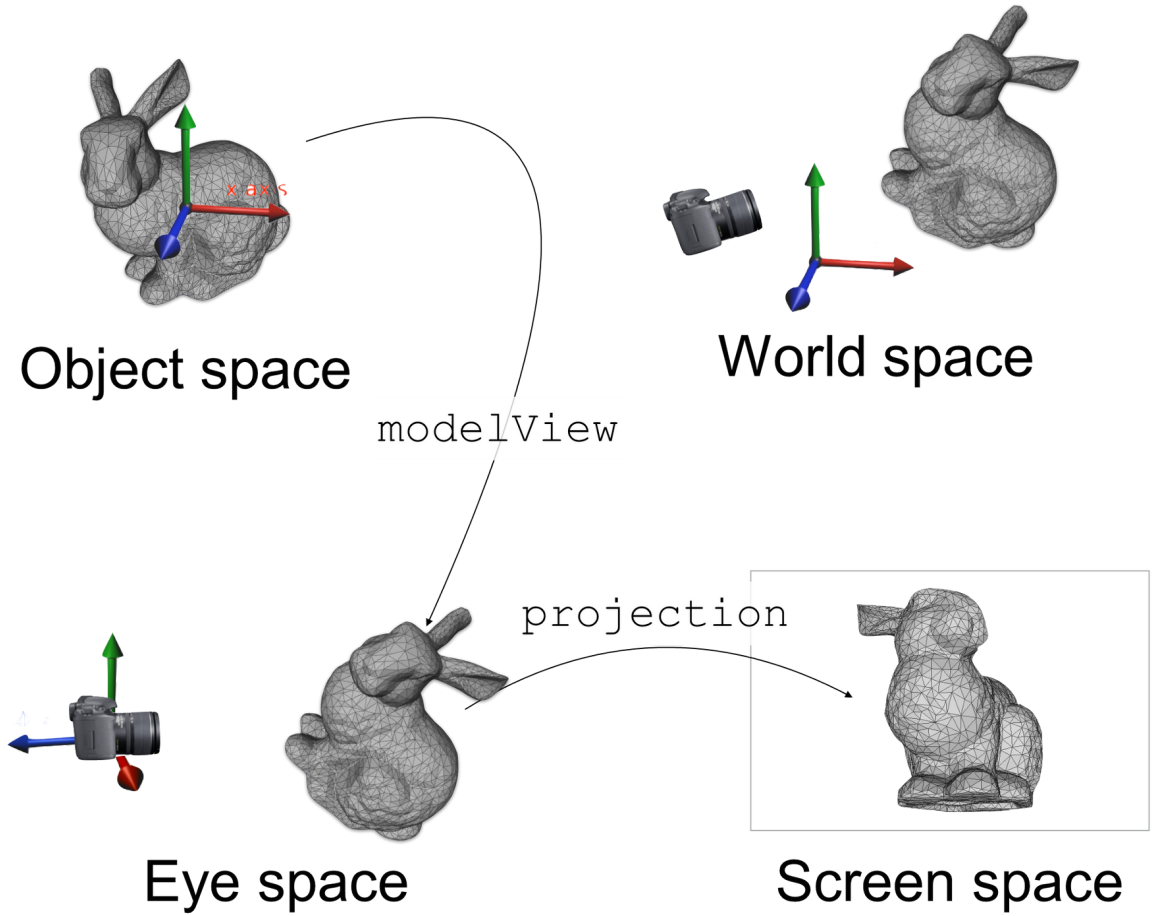
Screen space

Coordinate Spaces



$$\mathbf{p}_{screen} = \text{Projection} * \text{View} * \text{Model} * \mathbf{p}_{object}$$

Coordinate Spaces



$$\mathbf{p}_{screen} = Projection * View * Model * \mathbf{p}_{object}$$

Special Graphics Matrices

Model matrix: Translates, rotates, and/or scales each model (car, ball, ant, whatever) to transform vertices defined in *object space* into the desired positions in *world space*.

View matrix: Translates and/or rotates the entire world to take into account the current position and orientation of the camera. After applying the view matrix, the camera will be at the origin looking down the -Z axis, so the view matrix needs to translate/rotate the whole world into this arrangement. In this way, the view matrix transforms *world space* into *eye space*.

ModelView matrix: The combination of the two above matrices. Be careful about the correct ordering for the multiplication. Model gets applied first, then view, so the correct ordering is: view * model

Projection matrix: Transforms *eye space* to 2D *screen space*, taking perspective projection and aspect ratio of the window into account.

Special Graphics Matrices

Model matrix: Translates, rotates, and/or scales each model (car, ball, ant, whatever) to transform vertices defined in *object space* into the desired positions in *world space*.

View matrix: Translates and/or rotates the entire world to take into account the current position and orientation of the camera. After applying the view matrix, the camera will be at the origin looking down the -Z axis, so the view matrix needs to translate/rotate the whole world into this arrangement. In this way, the view matrix transforms *world space* into *eye space*.

ModelView matrix: The combination of the two above matrices. Be careful about the correct ordering for the multiplication. Model gets applied first, then view, so the correct ordering is: view * model

Projection matrix: Transforms *eye space* to 2D *screen space*, taking perspective projection and aspect ratio of the window into account.

Normal matrix: Same function as **Model matrix**, but for use with normals!

Transforming Normals to World Space

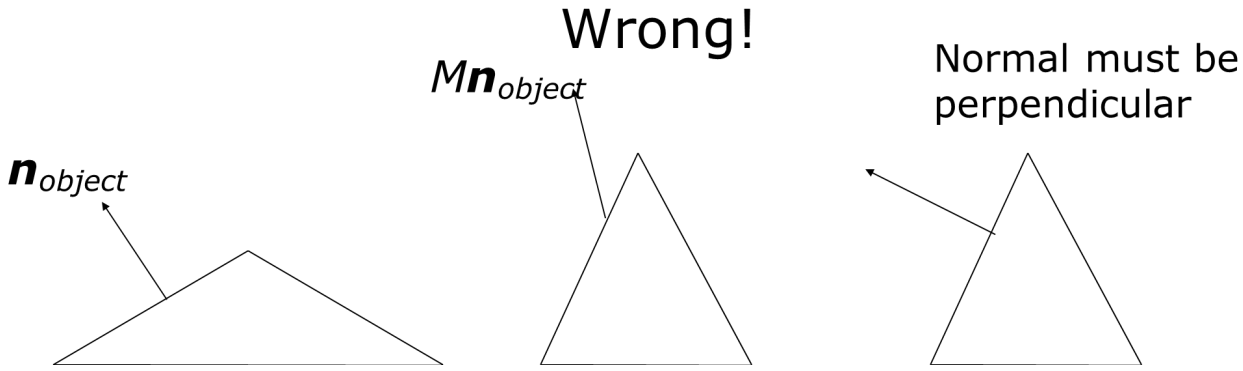
- Given a normal vector in *object space*
- We want a normal vector in *world space* for the lighting equation
- To transfer an object to world coordinates, we just multiplied its vertices by *M*
- Can we just treat the normal vector the same way?

Transforming Normals to World Space

- Given a normal vector in *object space*
- We want a normal vector in *world space* for the lighting equation
- To transfer an object to world coordinates, we just multiplied its vertices by *M*
- Can we just treat the normal vector the same way?

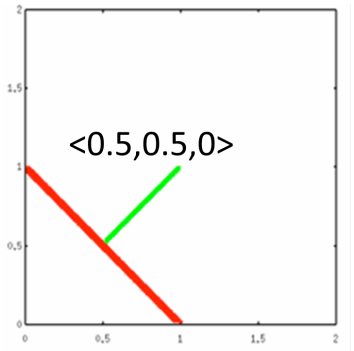
$$\boldsymbol{n}_{world} \neq M\boldsymbol{n}_{object}$$

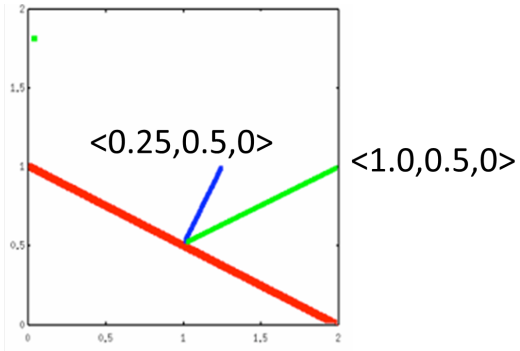
- Example: let's consider an M that scales x by .5 and y by 2



Transforming Normals to World Space

- Why doesn't it work to transform the normal by the same *M* that we use for the vertices?
- Well, actually it does work for translation, rotation, and **uniform** scales.
- **Non-uniform** scales cause a problem. The normal is distorted by the opposite of the scale applied to the surface.
- 2D example: scale *x* by 2.0 and *y* by 1.0.
- To get the correct normal, we need to invert the non-uniform scale.





Transforming Normals to World Space

- If not *M*, then what?
- Answer: $(\boldsymbol{M}^{-1})^{T}$
- Here is a hand-wavy explanation:

The problem is with non-uniform scales, so we want to apply the inverse of the non-uniform scale to the normal.

However, we want the rotation to remain the same as in M.

The translation doesn't matter because **n** is a vector, i.e. $\mathbf{n}_w=0$.

For the rotation part of ***M***, the transformed portion is equal to the original, i.e. $(\boldsymbol{R}^{-1})^{T} = \boldsymbol{R}$; the inverse reverses the rotation, but the transpose reverses it back to the original rotation.

For the scale part of ***M***, the inverse inverts the scale, but the transpose does nothing. Scale is only along the diagonal, which doesn't change during a transpose: $(\boldsymbol{S}(x,y,z)^{-1})^{T} = \boldsymbol{S}(x,y,z)^{-1} = \boldsymbol{S}(1/x, 1/y, 1/z)$

Using the Five Matrices in Assignment 5

You will not need to compute any of these matrices for assignment 5.

The model, view, projection, and normal matrices have already been calculated for you and sent to each shader.

Depending on the shader, sometimes the model and view matrices have been combined into the modelView matrix.

You will just need to figure out how the matrices are used in each shader.