Uniforms Cont. & 3D Matrices in GLSL & Normals

Game 300

James Dupuis

Objectives

- Learn about:
  - Uniforms
  - Viewports & Matrices through GLSL
  - Normal Data & other Math calculations for lighting

UNIFORM

- value is a **const** and cannot be changed from the GLSL code.
- Accessing in shader code:
  - **uniform float someValue;**
- With a uniform variable added to a shader you can call the following function inside the application code:
  - **GLint glGetUniformLocation(GLuint program, const char* name);**
- This function is used to retrieve a handleID to the uniform for access throughout the application
  - This takes in two parameters:
    - The **program object handleID** (GLuint)
    - The **name of the variable** used in the shader code. (char*)
  - Example:
    - **GLint someValUniformHandle;**
    - **someValUniformHandle = glGetUniformLocation(promgramObj, "someValue");**
  - OpenGL links the uniforms together with the program object when the **glLinkProgram()** function is called.
  - This means if you change the uniforms available to a shader or program at runtime, you need to relink the program object.

Setting uniforms from OpenGL

- With the handle available we can now set the uniform value into the shaders similar to an in into the vertex shader using an OpenGL API call:
  - **glUniform1f(handle, value);**
  - **handle** here is a handle for the specific uniform we retrieved previously.
  - **value** is the new value you would like to pass to the shaders.

- if a **uniform** is unused within the shader, it will be removed by the openGL compiler to optimize it.
  - If this is the case, the uniforms **handle will be -1**.

- Q: When do you use a **in** or **out** variable?
  - A: when you need to pass variables created from one shader to another.
    - When you need to modify variables based on the outcome of a subshader.
- Q: When should you use a **uniform** variable?
  - A: When you need to pass a variable to a specific shader or the same value to multiple shaders

Viewports/ Matrices in Shaders

- We touched on 3D math and how viewports can be setup for the old style OpenGL, but how does our shader code work with these values.
  - They don't.
  - Shaders require you to create and manage each matrix on your own.
    - These matrices will need to be passed into the Shaders for calculating and applying depth and transformation changes.
- Fortunately the **Vmath** library has many functions we can use to create our viewport and setup our matrices similarly to what we did through the OpenGL API.
  - Vmath::**Perspective**() or vmath::Ortho will create our viewport
  - Vmath::**Lookat**() will allow us to create a view matrix for our camera
- Once we have these matrices setup and/or modified, we will pass them in to our shaders using **Uniforms**.

Viewport

- To setup the Viewport matrix there are also vmath functions available to handle this for you and produce a matrix4.
  - Vmath::**perspective**(float fovy, float aspect, float n, float f);
    - Fovy is the field of view angle in degrees.
    - **Aspect ratio** is generally calculated using your window width / the window height.
    - **N** is the near clipping distance, how close to the camera position is displayed.
    - **F** is the far clipping distance, how far does the scene go.

```
static inline mat4 perspective(float fovy, float aspect, float n, float f)
{
    float q = 1.0f / tan(radians(0.5f * fovy));
    float A = q / aspect;
    float B = (n + f) / (n - f);
```

Vmath Perspective

```
static inline mat4 perspective(float fovy, float aspect, float n, float f)
{
    float q = 1.0f / tan(radians(0.5f * fovy));
    float A = q / aspect;
    float B = (n + f) / (n - f);
    float C = (2.0f * n * f) / (n - f);

    mat4 result;

    result[0] = vec4(A, 0.0f, 0.0f, 0.0f);
    result[1] = vec4(0.0f, q, 0.0f, 0.0f);
    result[2] = vec4(0.0f, 0.0f, B, -1.0f);
    result[3] = vec4(0.0f, 0.0f, C, 0.0f);

    return result;
}
```

Vmath Ortho

- If you desire an orthographic view, vmath also has a function to set that matrix up for you:
  - vmath::**ortho**(float left, float right, float bottom, float top, float near, float far);
    - Left, right, top and bottom are all float values indicating the views far positions -1.0f -> 1.0f
    - Near and far are similar to perspective which define the depth of the view.

```
static inline mat4 ortho(float left, float right, float bottom, float top, float n, float f)
{
    return mat4( vec4(2.0f / (right - left), 0.0f, 0.0f, 0.0f),
                 vec4(0.0f, 2.0f / (top - bottom), 0.0f, 0.0f),
                 vec4(0.0f, 0.0f, 2.0f / (n - f), 0.0f),
                 vec4((left + right) / (left - right), (bottom + top) / (bottom - top), (n + f) / (f - n), 1.0f) );
}
```

glGetUniformLocation

- We can now pass data for a matrix from the application to all shaders that make use of the uniform by the same name using **glUniformMatrix** calls:

```
render()
{
    glUniformMatrix4fv(mv_location, 1, GL_FALSE, mv_matrix);
}
```

- This takes 4 parameters:
  - handle: The handle for the uniform to set
  - Count : number of matrices (we'll always use 1 here)
  - transpose : For the matrix commands, specifies whether to transpose the matrix as the values are loaded into the uniform variable.
  - the variable to send, in this case our matrix.
- There are many uniform passing functions available similar to the VertexAttrib functions, depending on the variable type needed:
  - https://www.khronos.org/registry/OpenGL-Refpages/gl4/html/glUniform.xhtml

Using Vmath

```
void CameraManager::SetupCamera()
{
    CameraPosition.x = 0;
    CameraPosition.y = 0;
    CameraPosition.z = 0;

    static const GLfloat one = 1.0f;

    // GLint x, GLint y,    GLsizei width,  GLsizei height);
    glViewport(0, 0, WINDOW_WIDTH, WINDOW_HEIGHT);
    glClearBufferfv(GL_DEPTH, 0, &one);

    // Assign to a 4x4 matrix the projection or view of the world
    // similar to a camera system
    //          ________
    //        /|       /|
    //       / |     /  |
    //      /  |__/_____|
    //     /___/      /
    //     |    |   /
    //     |____|/
    //
    //                          fovy,                   aspect ratio,                   near clipping, far clipping
    proj_matrix = vmath::perspective(50.0f, (float)WINDOW_WIDTH / (float)WINDOW_HEIGHT,     1.0f,          20.0f);
}
```

We can setup a viewport using glViewport, then we create a perspective view on the viewport using the vmath::perspective() function.

The Params for this function are:

- Field of view range (y)
- Aspect ratio (width/height)
- Near clipping distance
- Far clipping distance

We now have a projection_matrix value as a mat4 variable available through the vmath library.

We then need to forward our Projection matrix into the shader code using uniforms.

Uniform Handles Applied

- Inside of our shader code we need to use our projection matrix, a modelview matrix, and the individual vertex positions to render an object.
- We can use the glGetUniformLocation function to search the program objects shader files for a uniform variable that matches the string supplied, just as we did for textures:

```
void ShaderManager::FindUniformHandles()
{
    TextureUniformHandle = glGetUniformLocation(programObj, "texture0");// Get the initial matrices references from the program.

    ModelViewUniformHandle = glGetUniformLocation(programObj, "mv_matrix");

    ProjectionUniformHandle = glGetUniformLocation(programObj, "proj_matrix");
}
```

- Before we render our object, we use this handle to forward data into that location of the shaders using glUniformMatrix4fv() passing in the matrix from the CameraManager into the handle Acquired above:

```
vmath::mat4 proj_matrix = CameraManager::GetInstance()->getModifiedProjectionMatrix();
glUniformMatrix4fv(ProjectionUniformHandle, 1, GL_FALSE, proj_matrix);
```

Into the Shader

- We can create a modelView matrix by creating a brand new mat4 (Matrix) variable using the vmath::translate() function.
    - This returns a full matrix with all translation modifications made.
- We can use the same process to pass our individual Models Matrix transformation changes to the shaders by using the glUniformMatrix4fv function again()

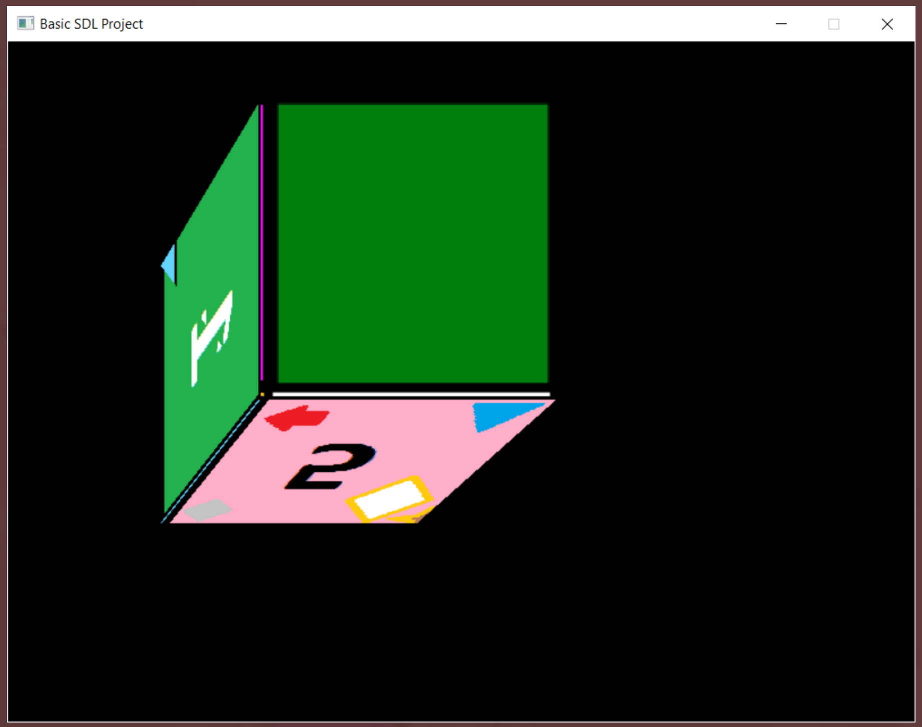
```
vmath::mat4 mv_matrix = vmath::translate(position.x, position.y, position.z);
glUniformMatrix4fv(ModelViewUniformHandle, 1, GL_FALSE, mv_matrix);
```

- Inside of the shader itself, we declare a uniform for both the mv_matrix & proj_matrix.
- This will be populated with the data from our glUniformMatrix4fv calls.
- We then use both of these matrices, multiplied together with the individual vertices positions to produce our three dimensional point.

```
#version 430 core

layout(location = 0) in vec3 VertPos;
layout(location = 1) in vec2 UVs;

uniform mat4 mv_matrix;
uniform mat4 proj_matrix;

out vec2 UV;

void main(void)
{
    UV = UVs;

    // new position using the projection and modelview matrices in addition to the position of each vertices
    vec4 P = proj_matrix * mv_matrix * vec4(VertPos, 1.0f);

    gl_Position = P;
```

3D Completed

- Our Local Data for our Matrices can now be used to create three dimensional rendering of objects at depths.
- We can also apply textures to our objects.
- Our final step towards realism and the true power of shaders comes from performing lighting calculations.



Dot Product

- Dot Product (2 Vectors)
  - Produces a single float value (scalar)
  - Cosine of the angle between the two vectors scaled by the product of their length:
    - V1•V2 = (x1 * x2) + (y1 * y2) + (z1 * z2)
  - float dotp = vmath::dot(vec1, vec2)

- Useful for:
  - calculating lighting angles
  - A.I. – within view of a character
  - Particle effects – distribution angle

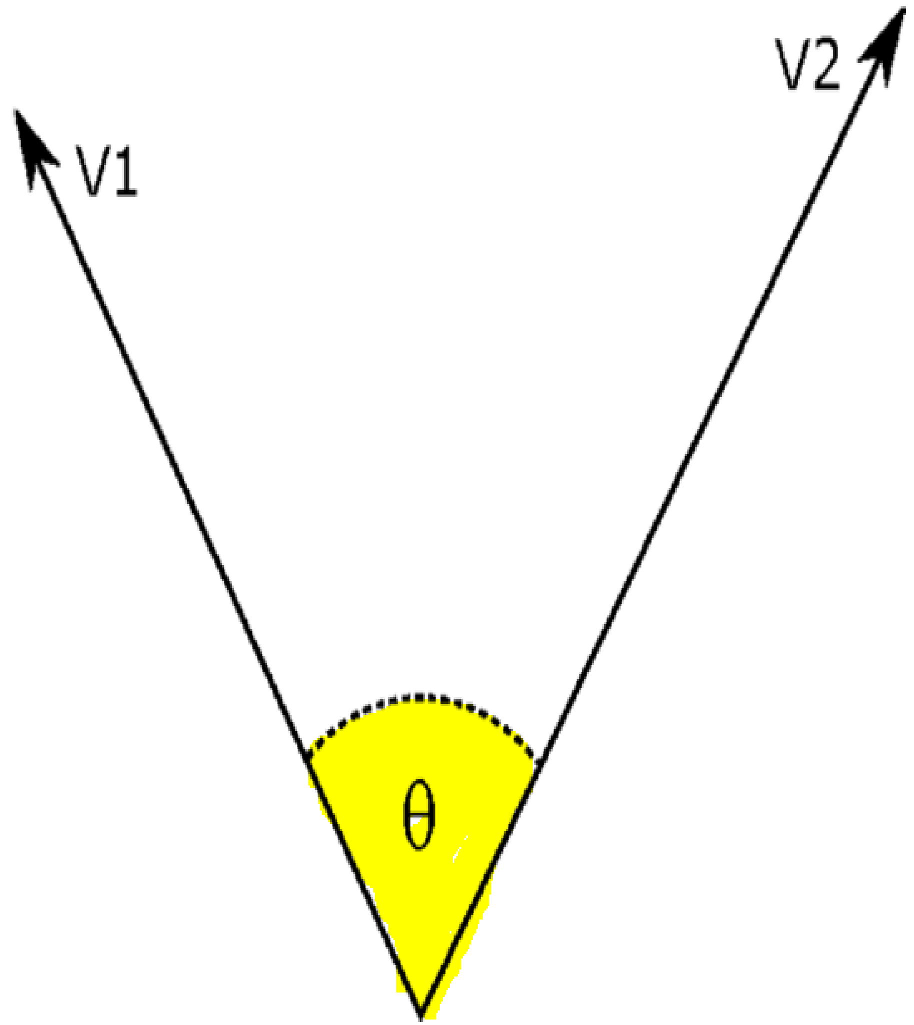


Figure 4.2: The dot product: cosine of the angle between two vectors

Dot Product (Vmath library)

```
template <typename T, int len>
static inline T dot(const vecN<T,len>& a, const vecN<T,len>& b)
{
    int n;
    T total = T(0);
    for (n = 0; n < len; n++)
    {
        total += a[n] * b[n];
    }
    return total;
}
```

Cross Product Example

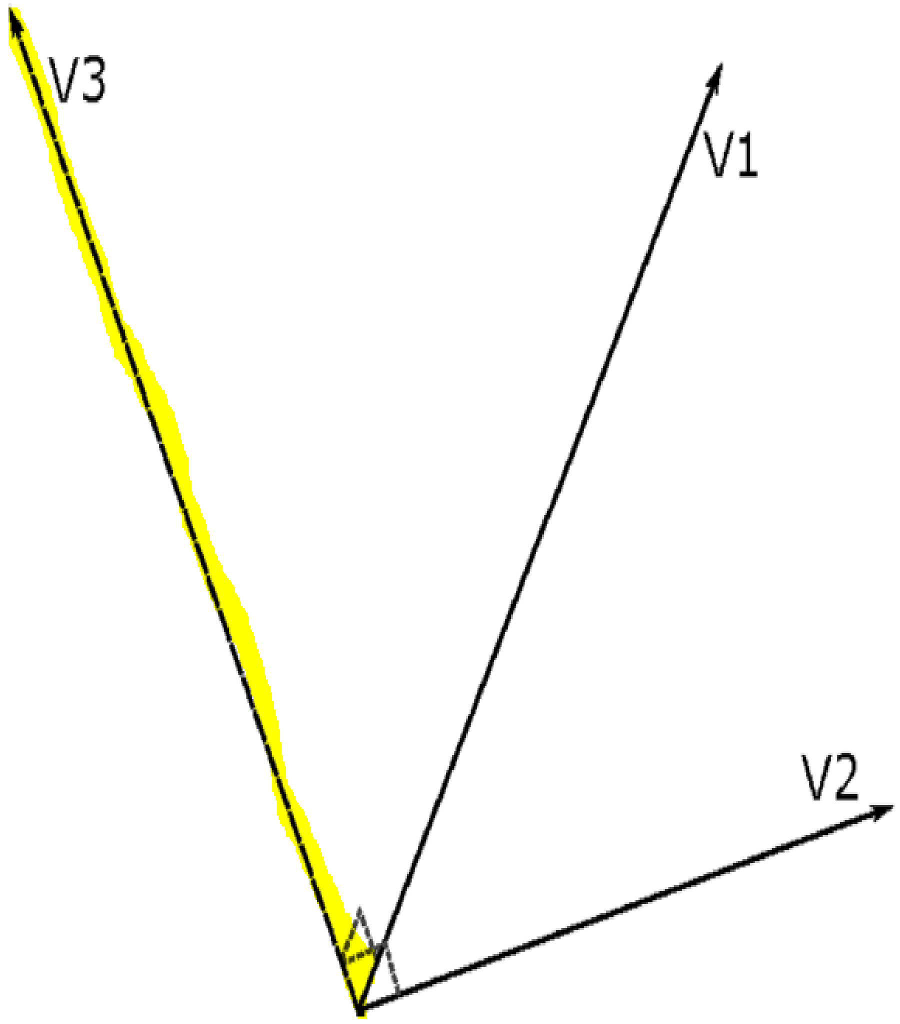
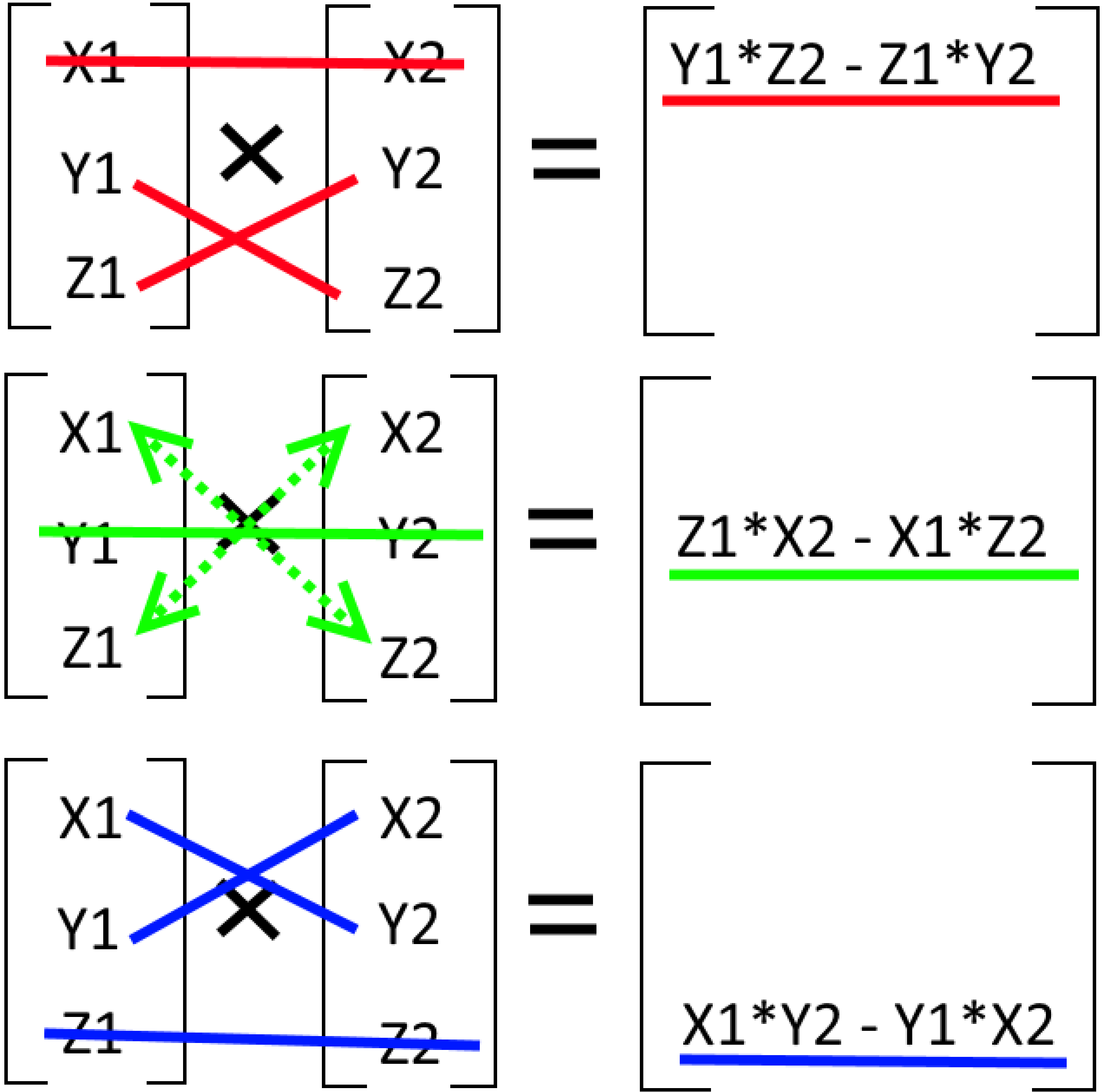


Figure 4.3: A cross product returns a vector perpendicular to its parameters

- Cross Product (2 Vectors)
  - Produces a new Vec3 perpendicular to the 2 vectors provided.
  - V1 x V2 = $\begin{vmatrix} V1x \\ V1y \\ V1z \end{vmatrix} \times \begin{vmatrix} V2x \\ V2y \\ v2z \end{vmatrix}$
    - Calculation rules on next slide
  - vmath::vec3 cp = vmath::cross(v1, v2);
  - Order of vectors is important for determining which way the new vector points.
  - Right hand rule.
- Used in lighting calculations to find normals of objects hit.

Cross Product Calculations



- Additional Aid / Tutorial:

  https://www.khanacademy.org/math/linear-algebra/vectors-and-spaces/dot-cross-products/v/linear-algebra-cross-product-introduction

- Vmath also has a function to calculate the cross product of 2 vectors.
  - Vec 3 a,b;
  - Vec3 cross = a.cross(b);

Cross Product Vmath

```
template <typename T>
static inline vecN<T,3> cross(const vecN<T,3>& a, const vecN<T,3>& b)
{
    return Tvec3<T>(a[1] * b[2] - b[1] * a[2],
                    a[2] * b[0] - b[2] * a[0],
                    a[0] * b[1] - b[0] * a[1]);
}
```

More vmath….

- To calculate the length or Magnitude of a vector, vmath has a handy function called length()
  - This essentially just calculates Pythagoras's theorem for you of :
    - Len = sqrt(x*x + y*y + z*z);

```
template <typename T, int len>
static inline T length(const vecN<T,len>& v)
{
    T result(0);

    for (int i = 0; i < v.size(); ++i)
    {
        result += v[i] * v[i];
    }

    return (T)sqrt(result);
}
```

More vmath….

- Lighting calculations will require calculating and analyzing how light is handled when it contacts materials and objects in the world.

```
template <typename T, const int S>
static inline vecN<T,S> reflect(const vecN<T,S>& I, const vecN<T,S>& N)
{
    return I - 2 * dot(N, I) * N;
}
```

  - Reflection is how the light bounces off of a surface
    - To calculate reflection vmath has a reflect() function.
  - Refraction is the angle at which the light may traverse through a transparent object like ice, water or a diamond.
    - To calculate refraction vmath has a refract() function

```
template <typename T, const int S>
static inline vecN<T,S> refract(const vecN<T,S>& I, const vecN<T,S>& N, T eta)
{
    T d = dot(N, I);
    T k = T(1) - eta * eta * (T(1) - d * d);
    if (k < 0.0)
    {
        return vecN<T,N>(0);
    }
    else
    {
        return eta * I - (eta * d + sqrt(k)) * N;
    }
}
```

Summary

- We can use glUniformMatrix4fv calls to pass information about projection and model matrices to our shaders for 3D Math calculations.
- Lighting requires calculations to determine normal of faces, reflection or refraction of light rays.
  - We can use the dot product to determine angles and the cross product to determine normals.
- Vmath has many useful functions to help us compute these values when necessary.