Lecture 05

3D Graphics

- Higher version of OpenGL and OpenGL ES
- Homogeneous coordinate system
- View control

Higher Version of OpenGL

- The latest version is OpenGL 4.5.
- A sub-set version for portable devices is called OpenGL ES. The latest version is OpenGL ES 3.0

Earlier Version: OpenGL 1.1

- Characterized by glBegin, glVertex, and glEnd.
- It is called immediate mode.
- A vertex can carry fixed information.
  - Coordinate
  - Normal
  - Color
  - Texture Coordinate
- A vertex goes through a fixed function pipeline.
  1. Model-view transformation
  2. Projection transformation
  3. Rasterization
  4. Depth test, alpha blending, texture sampling
  5. Written to the pixel.

- Limitations of the immediate mode
  - OpenGL cannot tell the number of incoming vertices until glEnd().
  - The driver can only make a guess of how many pipelines to run in parallel upfront. (Cannot optimize the resource allocation.)
  - A vertex may want to carry more information than coordinate, normal, color, and texture coordinate.
- Limitations of the fixed-function pipeline
  - It is fixed!
  - A texture can only be a color sample.
  - No freedom in how the vertex is transformed. (Very often want to apply some displacement between model-view and projection)
  - There are a lot of visual effects that can be achieved by tweaking the calculation in the rendering pipeline.

Solution: Programmable Shader

- OpenGL 2.0+ allows you to write your program for Vertex Shader and Fragment Shader.
- Vertex Shader
  Input: Vertex attributes
  Output: Projected vertex position and other variables for color calculation.
- Fragment Shader
  Input: Interpolated vertex position and other variables for color calculation
  Output: Pixel color

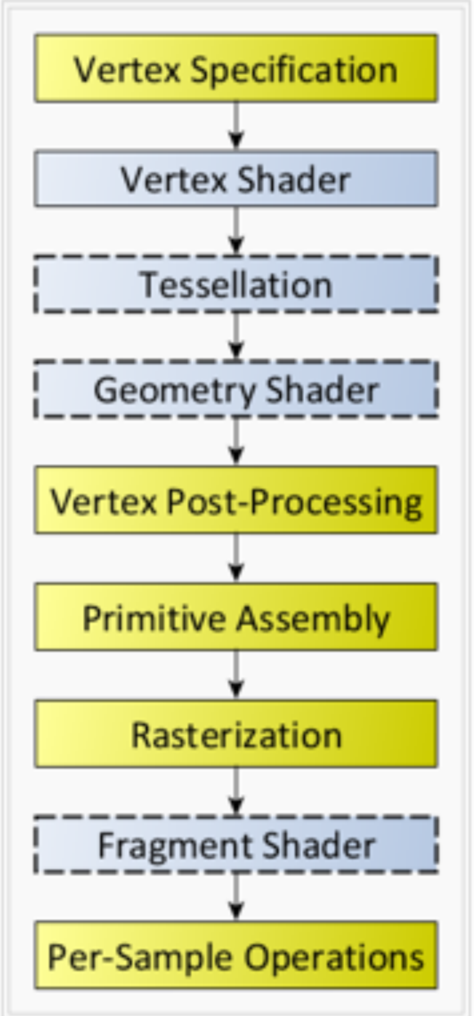


Diagram of the Rendering Pipeline. The blue boxes are programmable shader stages.

(*) Figure is from https://www.opengl.org/wiki/Rendering_Pipeline

Don’t be confused by the terminology

- Bad habit of computer scientists. They play with words!
- Shader: It doesn’t shade! It is just a PROGRAM!!!! (*)
- Fragment: It is just a PIXEL!!!! It is true that a pixel is a kind of fragment of a primitive, but you can understand better by calling it a pixel.

(*) In the OpenGL terminology, a rendering pipeline is a *program*, which consists of multiple computational units called *shaders*. A programmable rendering pipeline requires at least one vertex *shader* and fragment *shader*. OpenGL 2.x only allows you to write vertex- and fragment-shaders. Newer versions have more flexibility.

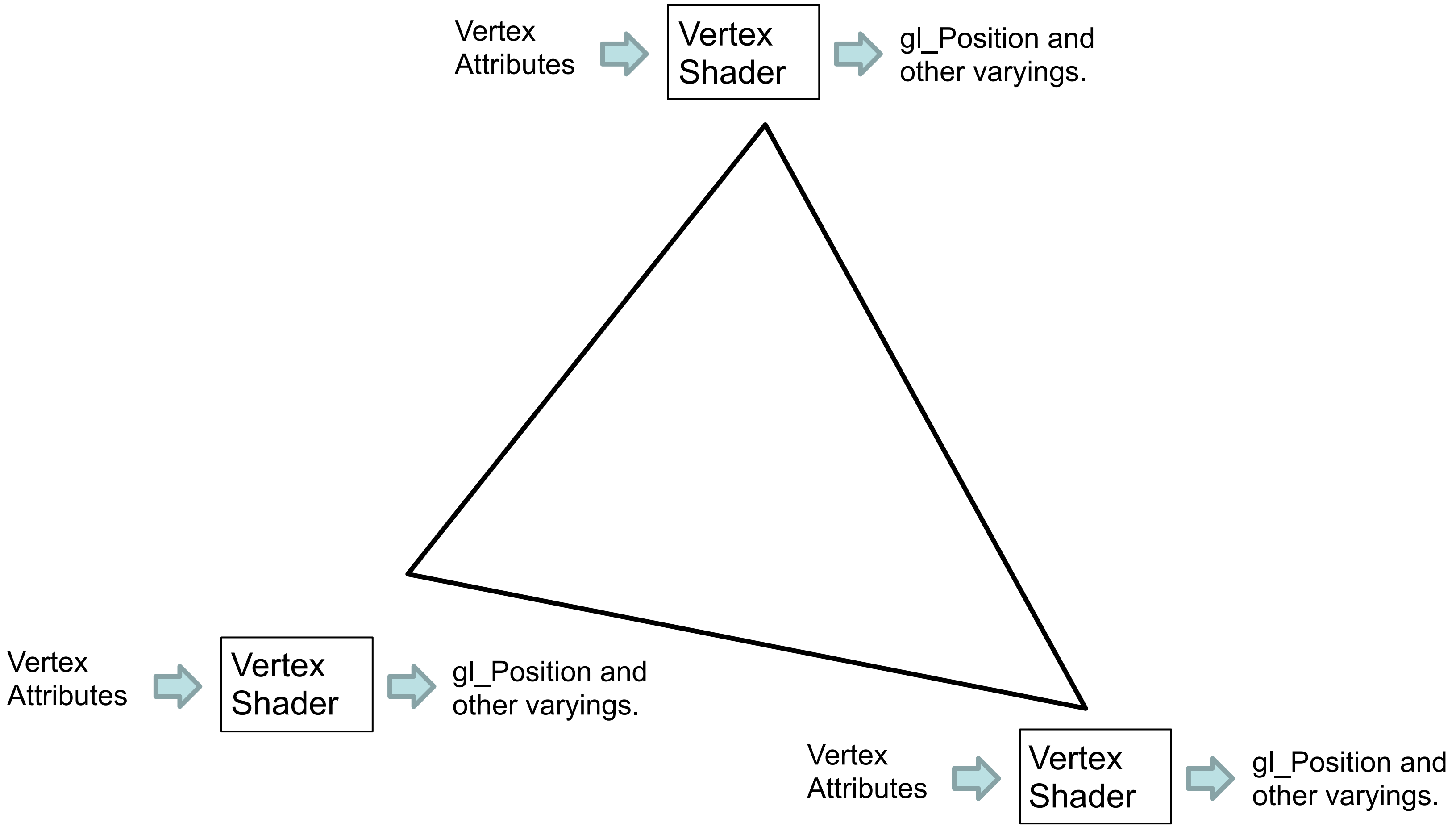
A vertex can carry more information

- A vertex can carry more or less information.
- The information that a vertex carries is called vertex attributes.
- Vertex attributes are defined in the vertex shader.

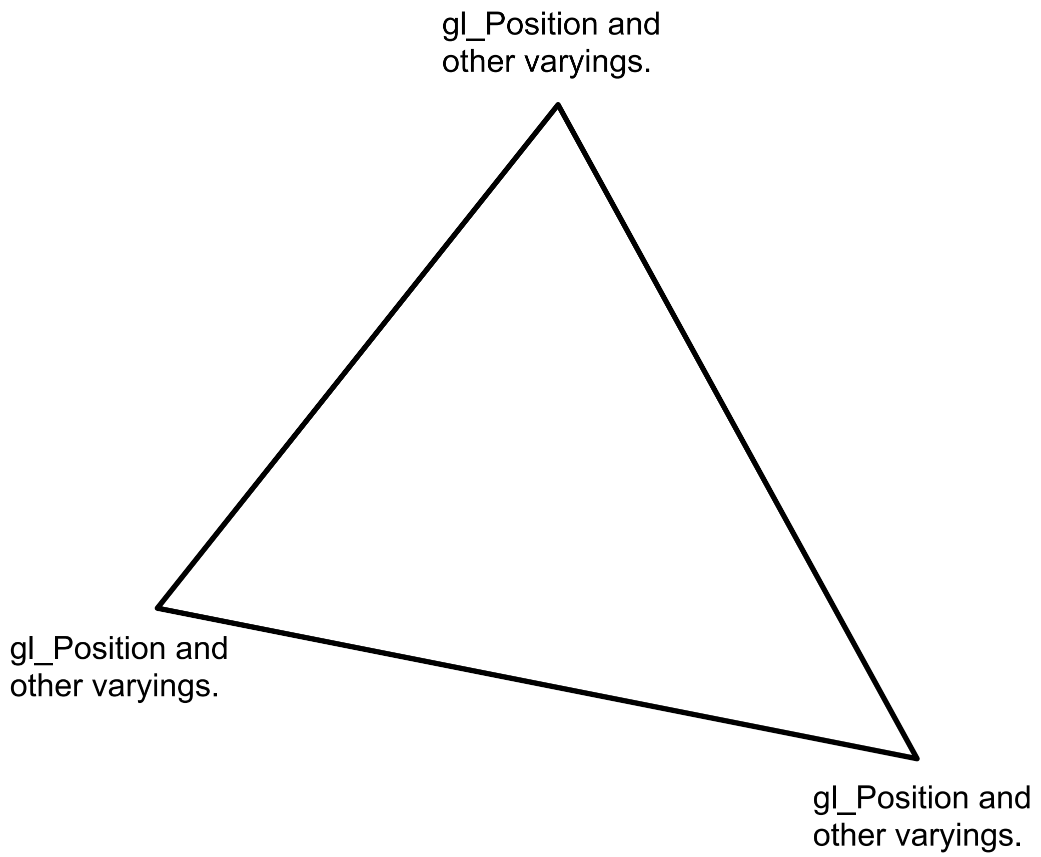
- The information that stays constant for a set of primitives is called uniform. For example, model-view matrix and projection matrix are uniforms.

Vertes-shader stage

For each vertex, the vertex shader calculates projected vertex position and other vertex variables called varying.

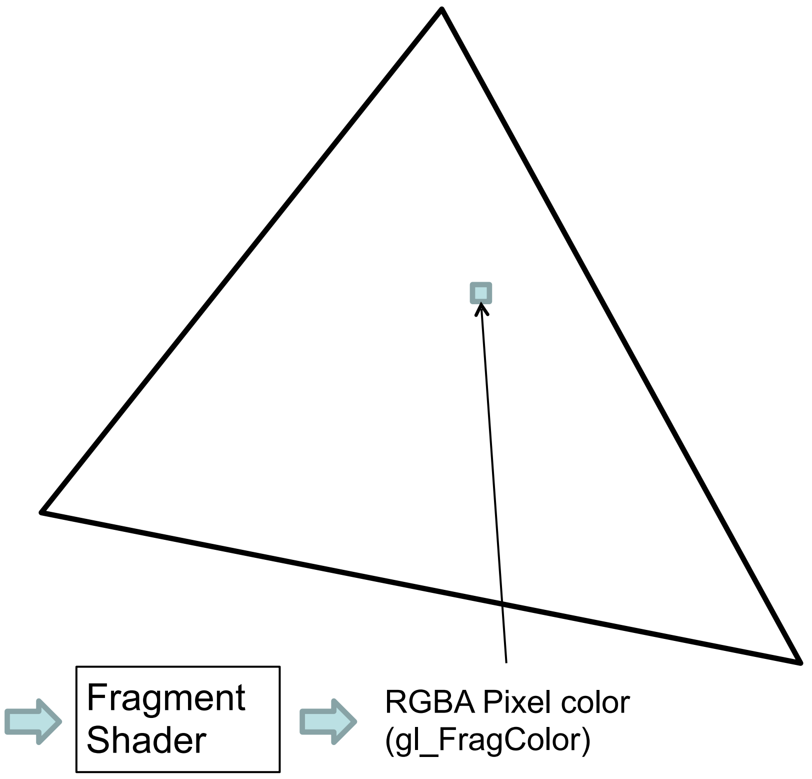

Vertex Attributes
Vertex Shader
gl_Position and other varyings.
Vertex Attributes
Vertex Shader
gl_Position and other varyings.
Vertex Attributes
Vertex Shader
gl_Position and other varyings.

After the vertex shader, each vertex has a projected position (gl_Position) and varyings.


gl_Position and other varyings.
gl_Position and other varyings.
gl_Position and other varyings.

Fragment-Shader stage

The rasterizer interpolates varyings for each pixel, and the interpolated varyings are given to the fragment shader. Finally the fragment shader decides the pixel value.



Advantages of the programmable shader

- It’s programmable! (Of course)
- You can pass only necessary vertex attributes. Can substantially reduce the CPU-GPU transaction.

Is programmable shader a solution? Or not?

Programmable shader also comes with some problems:

- GLSL programming. (Why can’t it be a C++ callback function?)
- Does OpenGL want to be a high-level rendering API, or low-level? (It would be a beautiful world if Direct 3D and Metal stays low-level, and OpenGL comes on top of a low-level API. But, OpenGL seems to be competing with them.)
- Graphics techniques have been fluid, and probably will be fluid for a while. We need to keep up.

Change in the program.

- Need to store all vertex attributes in an array. No glBegin-glEnd any more.
- Since the shader program can define its own transformation, no common matrices such as GL_PROJECTION and GL_MODELVIEW.
- Transformation matrices needs to be given as a uniform.

What you need to do to use a programmable shader?

- Do it once after OpenGL context is ready.
  - Create a program, vertex-shader, and fragment-shader identifiers.
  - Compile shader programs.
  - (Optional) Cache uniform and attribute IDs.
- When you draw something,
  - Call glUseProgram with the program identifier.
  - Set uniforms by glUniform.
  - Specify invariants (called generic attribute) by glAttrib.
  - Set up vertex-attribute arrays for other attributes.
  - Call glDrawArrays
  - Turn off vertex-attribute arrays

Let’s start with using a GLSL through a cover-up library.

1. Change directory to:
   (Your-user-directory)/eng_comp/src/(AndrewID)
2. Type:
   ```
   svn ../public/src/fslazywindow/template glsl_intro
   ```
3. Open CMakeLists.txt file in glsl_intro and modify TARGET_NAME and LIB_DEPENDENCY as:
   ```
   set(TARGET_NAME glsl_intro)
   set(LIB_DEPENDENCY fslazywindow ysclass ysglcpp ysglcpp_gl2)
   ```
4. Open top-level CMakeLists.txt and add the new project as:
   ```
   add_subdirectory(hummingbird/glsl_intro)
   ```

5. Re-run CMake. In Windows, you can change directory to the build directory and:

   ```
   cmake ../src -G “Visual Studio 12 Win64”
   ```

   Or in MacOSX, change directory to the build directory and:

   ```
   cmake ../src -G “Xcode”
   ```

   Or you can use cmake-gui.

6. Build and run glsl_intro.

   ```
   msbuild Project.sln /target:glsl_intro /p:Configuration=Release
   ```

Then modify the program.

- In main.cpp, add header files as:

```
#include <ysgl.h>
#include <ysglcpp.h>
#include <ysglslcpp.h>
```

- Populate Initialize and BeforeTerminate as:

```
/* virtual */ void FsLazyWindowApplication::Initialize(int argc,char *argv[])
{
	YsGLSLRenderer::CreateSharedRenderer();
}
/* virtual */ void FsLazyWindowApplication::BeforeTerminate(void)
{
	YsGLSLRenderer::DeleteSharedRenderer();
}
```

```
/* virtual */ void FsLazyWindowApplication::Draw(void)
{
    int wid,hei;
    FsGetWindowSize(wid,hei);

    glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
    {
        glViewport(0,0,wid,hei);

        YsGLSL2DRenderer renderer;  // Do not nest the renderer!
        renderer.UseWindowCoordinateTopLeftAsOrigin();

        GLfloat vtxCol[]=
        {
            0,0,
            (GLfloat)wid,0,
            (GLfloat)wid,(GLfloat)hei,
            0,(GLfloat)hei,

            0,0,1,1,
            1,0,0,1,
            0,1,0,1,
            1,0,1,1
        };
        renderer.DrawVtxCol(GL_TRIANGLE_FAN,4,vtxCol,vtxCol+8);
    }
    {
        GLfloat col[]={1,1,1,1};

        YsGLSLBitmapFontRendererClass renderer;
        renderer.RequestFontSize(24,24);
        renderer.SetViewportDimension(wid,hei);
        renderer.SetUniformColor(col);
        renderer.DrawString(100,100,"Test");
    }
    FsSwapBuffers();
    needRedraw=false;
}
```

- A renderer class stores:
  - Identifier of a shader program.
  - Identifiers of attributes and uniforms.
- The constructor of a renderer class does:
  - Get the shader program ready by calling glUseProgram.
  - Enabling vertex-attribute arrays by calling glEnableVertexAttribArray.
- DrawVtxCol function does:
  - Assign a vertex-attribute pointer by calling glVertexAttribPointer.
  - Draw primitives by glDrawArrays.

Going 3D

- Homogeneous coordinate system
- Projection transformation
- View transformation

Homogeneous coordinate system

Problem of the perspective projection

- $X_{screen}=X_{camer\_coord}*D_{prj\_plane\_dist}/Z_{camera\_coord}$
  $Y_{screen}=Y_{camer\_coord}*D_{prj\_plane\_dist}/Z_{camera\_coord}$
- Division by Z makes it a non-linear transformation.
- Everything else is linear.

- Instead of making it a special transformation, homogeneous coordinate system keeps it linear by increasing the dimension by one.

- Homogeneous coordinate

  (x,y,z,w)

  is equivalent to

  (x/w,y/w,z/w)
- It makes division-by-Z part in the fourth dimension.
- W=0.0 means infinitely long vector.

Projection Transformation

Perspective Projection

- More realistic than orthogonal projection.
- zNear and zFar will be transformed to z=-1.0 and z=1.0 in the projected window coordinate, respectively.
- W=-z in the projected window coordinate.

$$f = cotangent\left(\frac{fovy}{2}\right)$$

$$\begin{pmatrix} \frac{f}{aspect} & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & \frac{zFar+zNear}{zNear-zFar} & \frac{2\times zFar\times zNear}{zNear-zFar} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

(* matrix is from) https://www.opengl.org/sdk/docs/man2/xhtml/gluPerspective.xml

Orthogonal Projection

- Good for CAD/CAE applications.
- Z=zNear and Z=zFar are transformed to Z=-1.0 and Z=1.0 in the projected window coordinate, respectively.
- No division by Z. W=1.0 stays W=1.0.

$$\begin{pmatrix} \frac{2}{right-left} & 0 & 0 & t_x \\ 0 & \frac{2}{top-bottom} & 0 & t_y \\ 0 & 0 & \frac{-2}{farVal-nearVal} & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

where

$$t_x = -\frac{right+left}{right-left}$$

$$t_y = -\frac{top+bottom}{top-bottom}$$

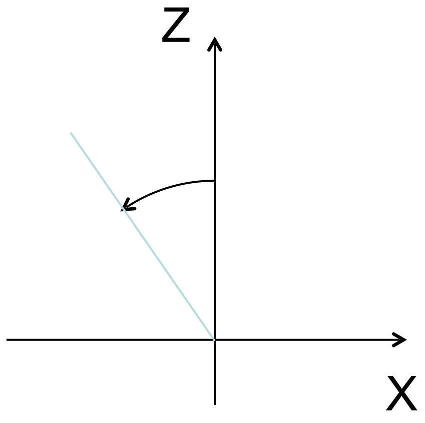
$$t_z = -\frac{farVal+nearVal}{farVal-nearVal}$$

(*) Matrix is from https://www.opengl.org/sdk/docs/man2/xhtml/glOrtho.xml

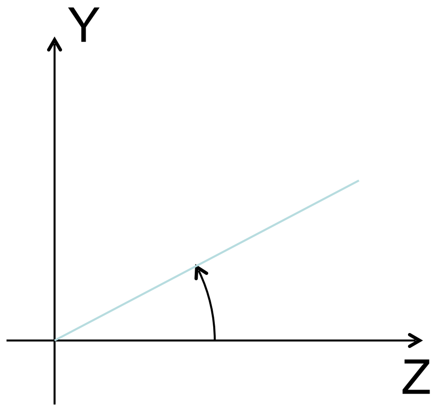
View transformation

- Minimum information for defining a viewer (camera) transformation:
  - Orientation (three angles)
  - Position (x,y,z)
- It is convenient to have a distance between the camera and the center of rotation.
  - Orientation (three angles)
  - Center of rotation (x,y,z)
  - Distance between the camera and the center of rotation

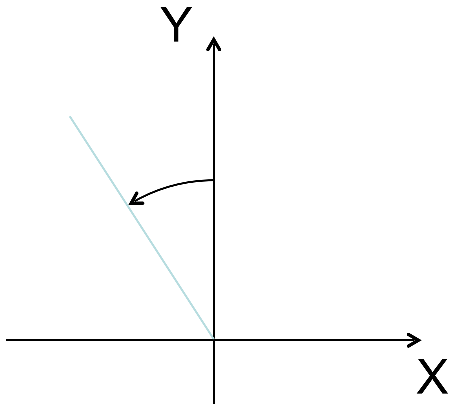
- Euler angle (h,p,b), or heading, pitch, and bank.
- To deal with both left-hand and right-hand coordinate system, these rotations are defined as:
  - Heading: Rotation on XZ plane.
  - Pitch: Rotation on ZY plane.
  - Bank: Rotation on XY plane.



Heading rotation



Pitch rotation



Bank rotation

- Then the rotation from the world coordinate to the camera coordinate can be written as:

$$R_{WtoC} = R_b^{-1} R_p^{-1} R_h^{-1}$$

- where:

$$R_h = \begin{pmatrix} \cos h & 0 & -\sin h & 0 \\ 0 & 1 & 0 & 0 \\ \sin h & 0 & \cos h & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}, \quad R_p = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos p & \sin p & 0 \\ 0 & -\sin p & \cos p & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix},$$

$$R_b = \begin{pmatrix} \cos b & -\sin b & 0 & 0 \\ \sin b & \cos b & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

-

- If rotation center is at (x_c, y_c, z_c) and the distance between the camera and the rotation center is d_c, the matrix that transforms a world-coordinate to the camera coordinate is written as:

$$T_{camera} = T_d R_{WtoC} T_C^{-1}$$

where:

$$T_d = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -d_C \\ 0 & 0 & 0 & 1 \end{pmatrix}, \quad T_C = \begin{pmatrix} 1 & 0 & 0 & x_c \\ 0 & 1 & 0 & y_c \\ 0 & 0 & 1 & z_c \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

One thing gets strange in the right-hand coordinate system

- Vector (0,0,1) is actually a backward vector.
- Forward vector is (0,0,-1)
- Positive p actually looks down.

YS-Class library

- I personally want to let you write your own vector- and geometry-calculation library, but the semester is too short.

- YS-Class library is a collection of functions and classes that I have written since I was an undergrad. (Open source with the BSD license.)
- May not be so popular, but used in several commercial programs.
- It provides with basic geometry calculations, matrix and vector classes, and some storage classes, etc.
- For the view-transformation, we use matrix and vector classes of YS-class library.

Implementing view transformation

1. Change directory to:
   (User Directory)/eng_comp/src/(AndrewID)
2. Type:
   ```
   svn export ../public/src/fslazywindow/template view_matrix
   ```
3. Change CMakeLists.txt as follows:
   ```
   set(TARGET_NAME glsl3d_view_matrix)
   set(LIB_DEPENDENCY fslazywindow ysclass ysgl ysglcpp ysglcpp_gl2)
   ```
4. Add headers in main.cpp
   ```
   #include <ysclass.h>
   #include <ysgl.h>
   #include <ysglcpp.h>
   #include <ysglslcpp.h>
   ```

5. Populate Initialize and BeforeTerminate functions as:

```
void FsLazyWindowApplication::Initialize(int argc,char *argv[])
{
	YsGLSLRenderer::CreateSharedRenderer();
}

void FsLazyWindowApplication::BeforeTerminate(void)
{
	YsGLSLRenderer::DeleteSharedRenderer();
}
```

6. Add member variables in FsLazyWindowApplication class:

```
	double h,p,b;
	YsVec3 viewTarget;
	double viewDist;
```

7. Initialize member variables in the constructor:

```
FsLazyWindowApplication::FsLazyWindowApplication()
{
        h=YsPi/4.0;
        p=YsPi/5.0;
        b=0;
        viewTarget.Set(0.0,0.0,0.0);
        viewDist=20.0;

        needRedraw=false;
}
```

8. Animate in the Interval function:

```
/* virtual */ void FsLazyWindowApplication::Interval(void)
{
   double t=(double)FsSubSecondTimer()/1000.0;
   h=YsPi*t;
   p=sin(YsPi*t)*YsPi/4.0;

   auto key=FsInkey();
   if(FSKEY_ESC==key)
   {
      SetMustTerminate(true);
   }
   needRedraw=true;
}
```

Add DrawPlainCube function

```
void FsLazyWindowApplication::DrawPlainCube(YsGLSLPlain3DRenderer &renderer,double d) const
{
    GLfloat df=(GLfloat)d;
    GLfloat vtxCol[]=
    {
        -df,-df,-df,  // 6 quads split into all triangles = 12 triangles
         df,-df,-df,  // 12 triangles times 3 vertices = 36 vertices
         df, df,-df,  // 36 vertices times 3 components = 108 floats

         df, df,-df,
        -df, df,-df,
        -df,-df,-df,

        -df,-df, df,
         df,-df, df,
         df, df, df,

         df, df, df,
        -df, df, df,
        -df,-df, df,

        -df,-df,-df,
         df,-df,-df,
         df,-df, df,

         df,-df, df,
        -df,-df, df,
        -df,-df,-df,

        -df, df,-df,
         df, df,-df,
         df, df, df,

         df, df, df,
        -df, df, df,
        -df, df,-df,

        -df,-df,-df,
        -df, df,-df,
        -df, df, df,

        -df, df, df,
        -df,-df, df,
        -df,-df,-df,

         df,-df,-df,
         df, df,-df,
         df, df, df,

         df, df, df,
         df,-df, df,
         df,-df,-df,

        1,0,0,1, 1,0,0,1, 1,0,0,1, 1,0,0,1, 1,0,0,1, 1,0,0,1, // Colors
        1,0,0,1, 1,0,0,1, 1,0,0,1, 1,0,0,1, 1,0,0,1, 1,0,0,1,
        0,1,0,1, 0,1,0,1, 0,1,0,1, 0,1,0,1, 0,1,0,1, 0,1,0,1,
        0,1,0,1, 0,1,0,1, 0,1,0,1, 0,1,0,1, 0,1,0,1, 0,1,0,1,
        0,0,1,1, 0,0,1,1, 0,0,1,1, 0,0,1,1, 0,0,1,1, 0,0,1,1,
        0,0,1,1, 0,0,1,1, 0,0,1,1, 0,0,1,1, 0,0,1,1, 0,0,1,1,
    };
    renderer.DrawVtxCol(GL_TRIANGLES,36,vtxCol,vtxCol+108);
}
```

It appears to be long, but it’s just drawing a cube in all triangles!

Populate Draw function

```
/* virtual */ void FsLazyWindowApplication::Draw(void)
{
    int wid,hei;
    FsGetWindowSize(wid,hei);

    glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
    glEnable(GL_DEPTH_TEST);

    YsProjectionTransformation proj;
    proj.SetProjectionMode(YsProjectionTransformation::PERSPECTIVE);
    proj.SetAspectRatio((double)wid/(double)hei);
    proj.SetFOVY(YsPi/4.0);
    proj.SetNearFar(0.1,100.0);

    GLfloat projMat[16];
    proj.GetProjectionMatrix().GetOpenGlCompatibleMatrix(projMat);


    YsMatrix4x4 view;
    view.Translate(0,0,-viewDist);
    view.RotateXY(-b);
    view.RotateZY(-p);
    view.RotateXZ(-h);
    view.Translate(-viewTarget);

    GLfloat viewMat[16];
    view.GetOpenGlCompatibleMatrix(viewMat);

    {
        YsGLSLPlain3DRenderer renderer;  // Again, do not nest the renderer!
        renderer.SetProjection(projMat);
        renderer.SetModelView(viewMat);
        DrawPlainCube(renderer,5.0);
    }

    FsSwapBuffers();

    needRedraw=false;
}
```

Add to SVN

- svn add view_matrix
- svn commit -m "Added view_matrix”

Lighting

What you need to specify:

- Light source (Light direction)
  - Light source is internally stored in the view (camera) coordinate system, because lighting is calculated in the view coordinate.
  - Since the same lighting condition is used for all vertices, light source is given as a uniform.
- Normal vectors
  - To calculate reflection intensity, a normal vector needs to be assigned to each vertex. A normal is an attribute because it can vary from vertex to vertex.

Lighting

Commonly-used lighting model - Four types of lights.

- Diffuse reflection
- Specular reflection
- Ambient
- Emission

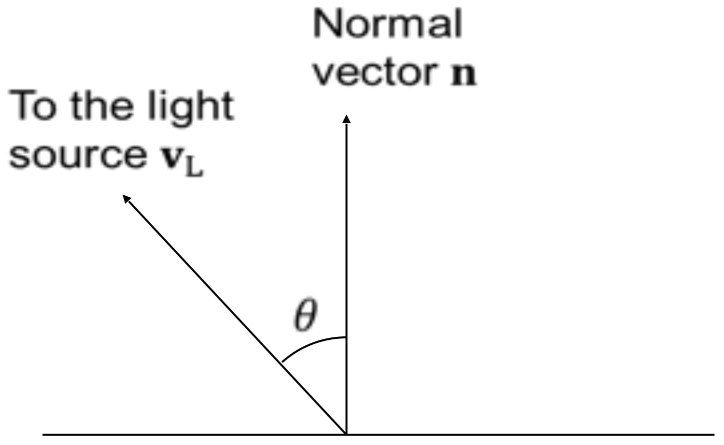
Lighting

- Diffuse reflection

When a light-ray hits a surface, some fraction of light is reflected uniformly to every direction, or diffused.

The reflection is brighter when the normal vector is closer to the vector to the light source.

$$c_{diffuse} = k_{diffuse}(\mathbf{v_L} \cdot \mathbf{n})$$

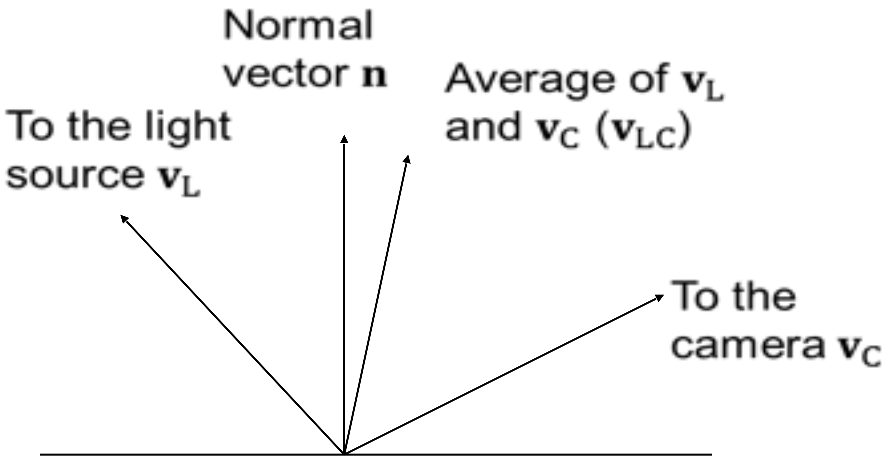

Normal vector $\mathbf{n}$
To the light source $\mathbf{v}_L$
θ

Lighting

- Specular reflection

Reflection like a mirror. The intensity is the highest when $\mathbf{v}_{\mathrm{LC}}$ is equal to $\mathbf{n}$. Also the high-intensity area should be small.

$$c_{specular} = k_{specular}(\mathbf{v_{LC}} \cdot \mathbf{n})^{specular\_exponent}$$


Normal vector $\mathbf{n}$
Average of $\mathbf{v}_{\mathrm{L}}$ and $\mathbf{v}_{\mathrm{C}}$ ($\mathbf{v}_{\mathrm{LC}}$)
To the light source $\mathbf{v}_{\mathrm{L}}$
To the camera $\mathbf{v}_{\mathrm{C}}$

Lighting

- Ambient light

In the bright environment, a primitive that is not directly facing the light source may not be completely dark due to reflections from surrounding objects.

- Emission

The color that the primitive is emitting.

1. svn copy view_matrix lighting
2. Edit CMakeLists.txt as:
   ```
   set(TARGET_NAME glsl3d_lighting)
   ```

Changes in the DrawPlainCube function.

```
void FsLazyWindowApplication::DrawPlainCube(YsGLSLShaded3DRenderer &renderer,double d) const
{
    GLfloat df=(GLfloat)d;
    GLfloat vtxCol[]=
    {
        -df,-df,-df,  // 6 quads split into all triangles = 12 triangles
         df,-df,-df,  // 12 triangles times 3 vertices = 36 vertices
         df, df,-df,  // 36 vertices times 3 components = 108 floats

         df, df,-df,
        -df, df,-df,
        -df,-df,-df,

        -df,-df, df,
         df,-df, df,
         df, df, df,

         df, df, df,
        -df, df, df,
        -df,-df, df,

        -df,-df,-df,
         df,-df,-df,
         df,-df, df,

         df,-df, df,
        -df,-df, df,
        -df,-df,-df,

        -df, df,-df,
         df, df,-df,
         df, df, df,

         df, df, df,
        -df, df, df,
        -df, df,-df,

        -df,-df,-df,
        -df, df,-df,
        -df, df, df,

        -df, df, df,
        -df,-df, df,
        -df,-df,-df,

         df,-df,-df,
         df, df,-df,
         df, df, df,

         df, df, df,
         df,-df, df,
         df,-df,-df,

        0,0,-1, 0,0,-1, 0,0,-1, 0,0,-1, 0,0,-1, 0,0,-1 // 36 vertices x 3 floats = 108 floats
        0,0, 1, 0,0, 1, 0,0, 1, 0,0, 1, 0,0, 1, 0,0, 1,
        0,-1,0, 0,-1,0, 0,-1,0, 0,-1,0, 0,-1,0, 0,-1,0,
        0, 1,0, 0, 1,0, 0, 1,0, 0, 1,0, 0, 1,0, 0, 1,0,
        -1,0,0, -1,0,0, -1,0,0, -1,0,0, -1,0,0, -1,0,0,
         1,0,0,  1,0,0,  1,0,0,  1,0,0,  1,0,0,  1,0,0,

        1,0,0,1, 1,0,0,1, 1,0,0,1, 1,0,0,1, 1,0,0,1, 1,0,0,1, // Colors
        1,0,0,1, 1,0,0,1, 1,0,0,1, 1,0,0,1, 1,0,0,1, 1,0,0,1,
        0,1,0,1, 0,1,0,1, 0,1,0,1, 0,1,0,1, 0,1,0,1, 0,1,0,1,
        0,1,0,1, 0,1,0,1, 0,1,0,1, 0,1,0,1, 0,1,0,1, 0,1,0,1,
        0,0,1,1, 0,0,1,1, 0,0,1,1, 0,0,1,1, 0,0,1,1, 0,0,1,1,
        0,0,1,1, 0,0,1,1, 0,0,1,1, 0,0,1,1, 0,0,1,1, 0,0,1,1,
    };
    renderer.DrawVtxNomCol(GL_TRIANGLES,36,vtxCol,vtxCol+108,vtxCol+216);
}
```

Changes in the Draw function

```
/* virtual */ void FsLazyWindowApplication::Draw(void)
{
    int wid,hei;
    FsGetWindowSize(wid,hei);

    glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
    glEnable(GL_DEPTH_TEST);

    YsProjectionTransformation proj;
    proj.SetProjectionMode(YsProjectionTransformation::PERSPECTIVE);
    proj.SetAspectRatio((double)wid/(double)hei);
    proj.SetFOVY(YsPi/4.0);
    proj.SetNearFar(0.1,100.0);

    GLfloat projMat[16];
    proj.GetProjectionMatrix().GetOpenGlCompatibleMatrix(projMat);

    YsMatrix4x4 view;
    view.Translate(0,0,-viewDist);
    view.RotateXY(-b);
    view.RotateZY(-p);
    view.RotateXZ(-h);
    view.Translate(-viewTarget);

    GLfloat viewMat[16];
    view.GetOpenGlCompatibleMatrix(viewMat);

    {
        GLfloat lightDir[]={0,0,1};

        YsGLSLShaded3DRenderer renderer;  // Again, do not nest the renderer!
        renderer.SetProjection(projMat);
        renderer.SetModelView(viewMat);
        renderer.SetLightDirectionInCameraCoordinate(0,lightDir);
        DrawPlainCube(renderer,5.0);
    }

    FsSwapBuffers();

    needRedraw=false;
}
```

Gouraud shading vs. Phong shading

- Gouraud shading

Lighting is calculated per vertex. The rasterizer interpolates the color within the primitive.

The lighting model of OpenGL 1.1. Faster than Phong shading.

- Phong shading

The rasterizer interpolates the normal vector within the primitive, and lighting is calculated per pixel.

Possible with the programmable shader.

Better rendering.

Slower than Gouraud shading because more varyings need to be interpolated.

Can see the difference by replacing YsGLSLShaded3DRenderer with YsGLSLPerVertexShaded3DRenderer.