3D Graphics Workshop

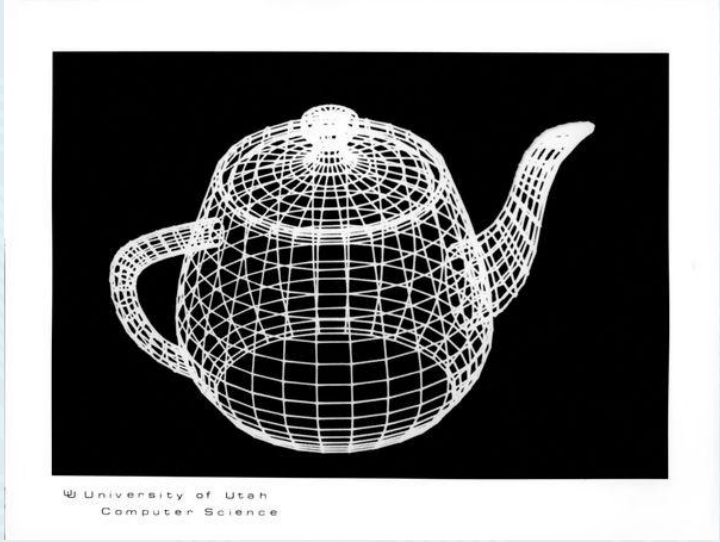


Image Source: https://www.computerhistory.org/revolution/computer-graphics-music-and-art/15/206

Led by

Ethan Balcik

2/4/2021

Introduction to 3D Graphics:

The Mathematics and History of 3D Computer Graphics

A Brief History of Computer Graphics

1940s - 1970s

- Late 1930s and early 1940s saw research into accurate lighting models
- Meanwhile, in the 1950s, the first computer-aided design (CAD) systems were being researched at companies like Boeing, General Motors, and Bell Labs
- In the 1960s, CAD and other computer graphics applications begin to take off within corporations and governmental sectors
  - In fact, William Fetter of Boeing coined the term computer graphics during this time
- MIT's Spacewar was the first arcade-style game, utilizing an analog CRT display
  - Undoubtedly influenced the development of arcade machines, which wouldn't take off commercially for another 10-20 years later
- Ivan Sutherland revolutionizes the field of computer graphics in the late 1960s, creating ten algorithms which solved many of the field's fundamental problems
- PONG is created by the soon-to-be founder of Atari games, Nolan Bushnell
- The Utah Teapot is created in 1975 by Martin Newell of the University of Utah (pictured on the title slide)

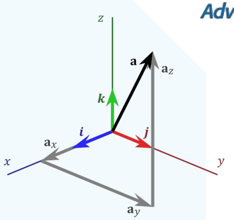
A Brief History of Computer Graphics

1970s - 1990s (and onward)

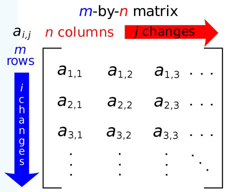
- Developments in abstracted & portable code took place in the mid 1970s
- Late 1979, Jim Clark designs a “Geometry Engine”, which later evolves into his company Silicon Graphics
- Advanced consumer computing technology utilizing computer graphics emerges in the early 1980s under brands like Microsoft and Apple
  - Introduction of the Graphical User Interface occurs during this time
- Various films utilizing CGI (Computer Generated Imagery) are released throughout the 1980s
  - PIXAR was one company that emerged during this time, which most notably developed heavily sophisticated shader technology utilized in Toy Story
- George Lucas’ Industrial Light & Magic furthered CGI to new highs in the 1990s
- OpenGL introduced in 1992 by Silicon Graphics
- 1990s and onward saw the development & advancement of the GPU, and saw faster innovation in 3D graphics in gaming and multimedia than ever before

Mathematical Concepts for Computer Graphics

Vectors, Matrices, & a Basic Overview of Relevant Linear Algebraic Concepts



- A **vector** is a list of n numbers specifying the coordinates of a point in an n-dimensional cartesian space
- A **matrix** is a table of m-by-n numbers specifying a transformation of n-dimensional cartesian space
- **Scaling** multiplying a vector by a single constant number, thereby scaling the magnitude of the vector by that constant
- **Vector addition** is the act of adding one vector's coordinates to those of another
- **Matrix-vector multiplication** is an operation that applies a transformation defined by a matrix to a vector
- **Matrix multiplication** the act of combining many transformations into one
- **Cross product** in 3-dimensional space takes in two 3D vectors and returns a vector perpendicular to those passed in (this is the right-hand-rule! :D)
- And a *TON* more math if you really want to get into computer graphics!



$$\begin{bmatrix} a & b \\ c & d \end{bmatrix} \cdot \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} ax+by \\ cx+dy \end{bmatrix}$$

$$\begin{bmatrix} a & b & c \\ d & e & f \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} = \begin{bmatrix} ax+by+c \\ dx+ey+f \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} ax+by+cz \\ dx+ey+fz \\ gx+hy+iz \end{bmatrix}$$

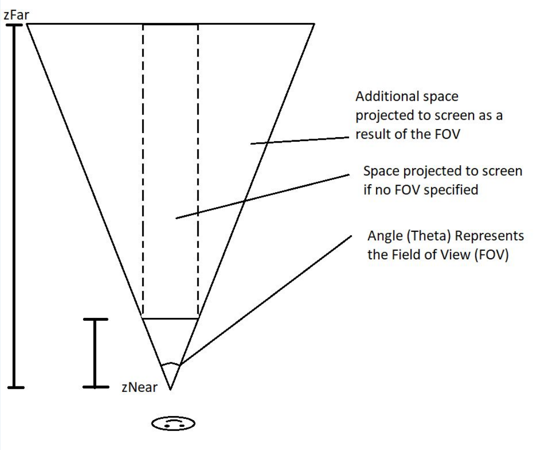
$$\begin{bmatrix} a & b & c & d \\ e & f & g & h \\ i & j & k & l \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} ax+by+cz+d \\ ex+fy+gz+h \\ ix+jy+kz+l \\ 1 \end{bmatrix}$$

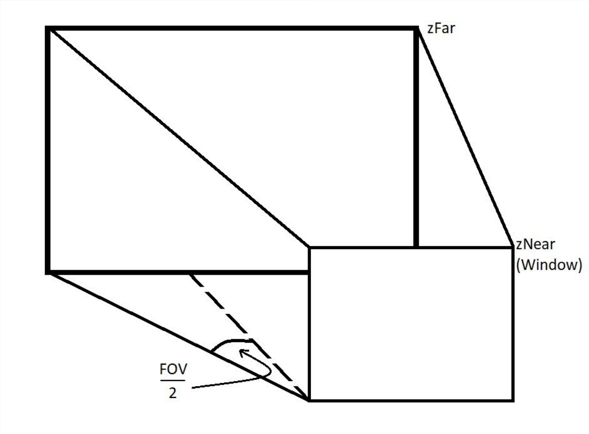
Image Sources: http://www.c-jump.com/bcc/common/Talk3/Math/Matrices/W01_0050_matrix_by_vector_mult.htm, https://upload.wikimedia.org/wikipedia/commons/thumb/6/6e/Cross_product.gif/330px-Cross_product.gif, https://upload.wikimedia.org/wikipedia/commons/thumb/b/bb/Matrix.svg/371px-Matrix.svg.png, http://www.chrisevans3d.com/pub_blog/maya-python-vector-math-primer/

Mathematical Concepts for Computer Graphics

Projection to a Plane, or "Camera" (1 of 2)

- Our window is our plane onto which we are projecting 3D objects
- We do so using a couple of important numbers
  - Aspect ratio = window height / window width, normalizes our window coords
  - Field of view, an angle measurement of our peripheral limits
  - Depth between viewer and window
  - Maximum depth of the space, measured relative to the viewer





Mathematical Concepts for Computer Graphics

Projection to a Plane, or "Camera" (2 of 2)

- The perspective projection matrix is derived using the numbers described previously, and each element adds its own set of desired properties
  - Element [0, 0] - Aspect ratio normalizes coordinates for the window, while the tangent expression results in the property of "zooming in" and "zooming out" as the field of view increases and decreases
  - Element [1, 1] - Same thing, except only concerned with the zooming property mentioned above
  - Elements [2, 2], [2, 3] - Normalize the z-coordinate
  - But wait, why is it 4D if we are operating in 3D?
    - 3D vectors append an additional z-value, making it a 4D vector for the purpose of taking the z coordinate and using it to relate the x and y coordinates inversely to its depth
      - Further = less motion on screen, closer = more
    - Element [3, 2] stores the z value in camera space

$$\begin{pmatrix} \dfrac{1}{ar \cdot Z \cdot \tan(\frac{\alpha}{2})} & 0 & 0 & 0 \\ 0 & \dfrac{1}{Z \cdot \tan(\frac{\alpha}{2})} & 0 & 0 \\ 0 & 0 & \dfrac{-FarZ}{NearZ - FarZ} & \dfrac{FarZ \cdot NearZ}{NearZ - FarZ} \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

Image Source: http://ogldev.atspace.org/www/tutorial12/12_11.png

Mathematical Concepts for Computer Graphics

Rotation and Translation of the "Camera" (1 of 2)

- We can imagine the viewer (or "Camera") as positioned in a larger space
  - The camera itself is defined by 3 vectors
    - Position vector
    - Up vector (Assume this is, by default, [0, 1, 0])
    - Forward vector (Assume this is, by default, [0, 0, 1])
  - Rotating the camera up and down involves transforming the up vector's y and z coordinates by trigonometric functions
  - Rotating the camera left and right involves transforming the forward vector's x and z coordinates by trigonometric functions, *and* transforming its y coordinate by the up vector's z coordinate transformation
    - The two must remain perpendicular (you'll see why next slide)

Mathematical Concepts for Computer Graphics

Rotation and Translation of the "Camera" (2 of 2)

- We can imagine the viewer (or "Camera") as positioned in a larger space
  - The camera itself is defined by 3 vectors
    - Position vector
    - Up vector (Assume this is, by default, [0, 1, 0])
    - Forward vector (Assume this is, by default, [0, 0, 1])
  - Moving the camera forward and backward involve simply adding a scaled forward vector to the position vector
  - Moving the camera left and right involves taking the cross product of the up and forward vector to receive a third vector which is perpendicular to both, and *then* adding this scaled vector to the position vector

OpenGL, and Other 3D Graphics Libraries:

Various Open Source & Industry Standard 3D Computer Graphics Libraries

The Khronos Group

OpenGL, WebGL, OpenGL ES, Vulkan

- OpenGL is an open source, cross-platform graphics API
- Developed by Silicon Graphics, inc., and has since been managed and expanded by the Khronos Group
- Its applications include
  - Computer-Aided Design (CAD)
  - Virtual Reality & Flight Simulation
  - Scientific Visualization
  - Video Games
- OpenGL ES is a well-defined subset of OpenGL desktop intended for use in low-power, embedded systems
  - WebGL is a cross-platform, royalty free web standard able to be recognized by OpenGL ES
  - WebGL brings plugin-free 3D to the web, implemented right in the browser
- Vulkan is a next-generation, low-level graphics API, designed for more balanced CPU-GPU usage -- used by Steam for compatibility mode (still in prototype)

Other Graphics Libraries

Commercial & Internal Graphics Libraries

- Microsoft DirectX - Direct3D
  - The ‘X’ in DirectX was used as the basis of the name “Xbox”
- RenderMan Interface Specification (RISpec)
  - Developed by Pixar Animation Studios for rendering photorealistic 3D scenes
- RenderWare
  - Developed before/during the emergence of GPUs
  - Once used for games able to be run on consoles including
    - GameCube
    - Wii
    - Xbox & Xbox 360
    - PlayStation 2 & PlayStation 3

OpenGL 3D Graphics Concepts

OpenGL Objects, Data Types, and Other Concepts (1 of 2)

- A **vertex** is a 3D vector, or a list of 3 points specifying coordinates in space
- A **vertex array object (VAO)** is a list of vertices, specifying the primitives of a “mesh”
  - **Primitives** in OpenGL include triangles, triangle strips, triangle fans, quads, quad strips, and polygons. In this workshop, we focus only on triangles.
- A **vertex buffer object (VBO)** stores a VAO, but as unformatted memory allocated by the GPU
- **Double Buffering** is the act of drawing to one buffer, while displaying another
  - Think of a buffer as a frame -- at any given time, there is one frame being drawn to, and one being displayed
  - When using double buffering, each time step the frames are switched.
- A **mesh** is a common way to refer to a set of primitives conceptually

OpenGL 3D Graphics Concepts

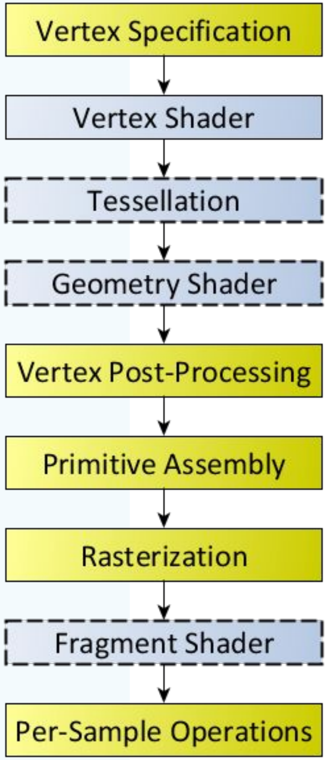
OpenGL Objects, Data Types, and Other Concepts (2 of 2)

- A **shader** is a program that takes in vertex data, draws it to the screen, and colors it on the screen
  - In OpenGL, shaders are written in GLSL, a C-like OpenGL language
- A **Vertex Shader** actually does the job of projection (and other transformations in space) of vertices
- A **Geometry Shader** takes simple primitives and transforms them as it sees fit
- A **Fragment Shader** adds color and shading to “fragments”, or assembled primitives using vertex data

OpenGL 3D Graphics Concepts

The OpenGL Graphics Pipeline

- VAOs/VBOs created by the programmer specify the vertices of a "Mesh"
  - Each vertex is processed by a Vertex Shader, also written by the programmer
  - Vertices are subject to optional tessellation and geometry shading
- Vertex Post-Processing, Primitive Assembly, and Rasterization handled automatically by OpenGL
- Rasterized Primitives are sent to the Fragment Shader for shading
  - At this stage, the fragment shader adds color to each of the fragments passed in
- Further operations handled automatically by OpenGL



3D Graphics Programming in OpenGL:

An Object-Oriented Approach to 3D Computer Graphics Engine Programming

Object Oriented Programming Refresher

Simple Refresher on Object Oriented Programming Concepts (1 of 2)

- A **class** is essentially a template for how data is formatted, similar to a struct in C, but more powerful and robust
  - A class can have **variables** (AKA, **fields**, **properties**, **members**), and **functions** (AKA, **methods**)
- An **object** is an “instance” of a class
  - A class is used as a blueprint for the allocation of actual specified data
- A **constructor** is a method defined in a class that creates the object & may initialize some of its properties
- A **destructor** is a method defined in a class that destroys the object
- A **getter** is a public method defined in a class that allows the programmer to access an object’s private properties when actually working with it
- A **setter** is a public method defined in a class that allows the programmer to change an object’s private properties when actually working with it

Object Oriented Programming Refresher

Simple Refresher on Object Oriented Programming Concepts (2 of 2)

- A **public variable/method** of an object can be accessed freely throughout the application whenever that object is in scope
- A **private variable/method** of an object can only be accessed within its class
  - Part of the Object Oriented Programming philosophy dictates that variables, for the most part, should be declared as private, and only accessed through an object's "interface", or its methods
- A **static variable/method** is a variable or method in a class to which all instances of that class share a common reference
  - I like to think of static variables & methods as being stored in the class itself, rather than the instances of that class

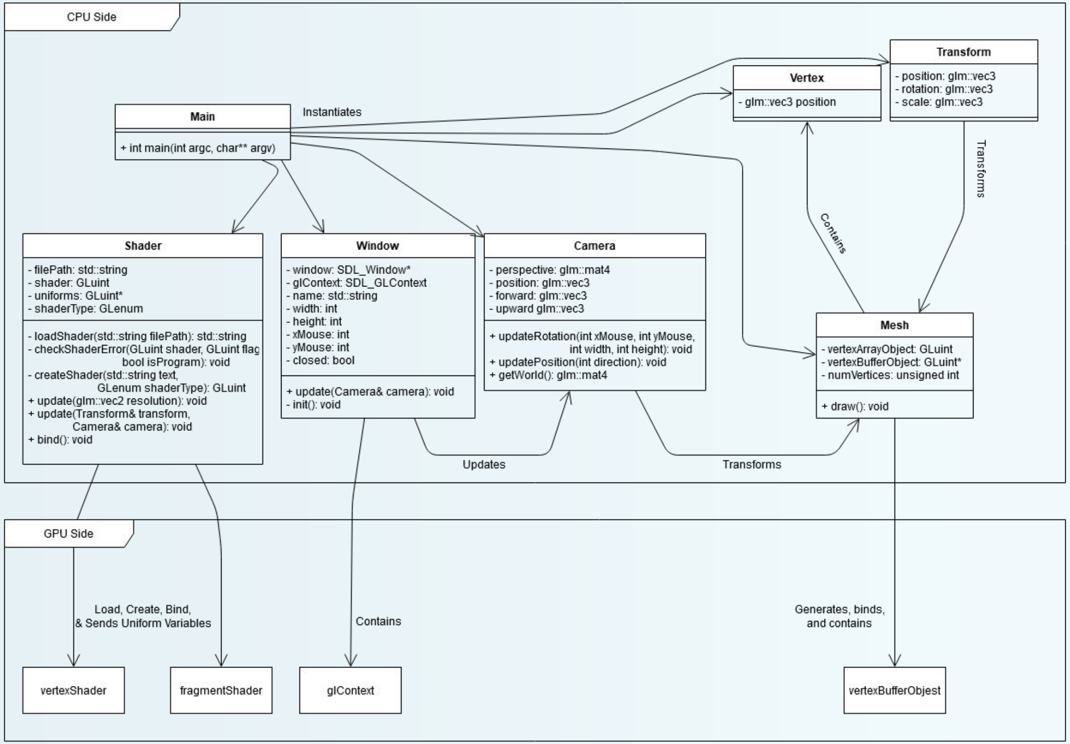
Unified Modelling Language (UML)

The Industry Standard for Visualizing an Object-Oriented Program

- UML is not a programming language, but it is a standard way to visualize complex object-oriented programs
- It communicates how classes and objects interact at a higher level than is easily detectable when analyzing source code alone
- In UML, classes & objects are represented as boxes, and relationships between classes are represented as arrows
- We will analyze what an Object-Oriented approach to writing a graphics engine might look like using a UML diagram

An Object-Oriented Graphics Engine

Example of an Object-Oriented Graphics Engine Visualized Using UML



- '-' = private
- '+' = public
- Boxes with attributes = classes
  - Main technically isn't a class, but it helps to visualize it as such
- Boxes with no attributes = objects & other data
- Arrows = relationships

Analyzing the Source Code

The Main Loop

```
//Main loop
while (!window.isClosed())
{
    //Clear the screen (floats below specify the clear color, I opted to use dark blue)
    glClearColor(0.09f, 0.13f, 0.4f, 0.0f);
    glClear(GL_COLOR_BUFFER_BIT);

    //Rotate the mesh we created around its y axis
    transform.setRotation(glm::vec3(transform.getRotation().x, transformCounter, transform.getRotation().z));

    //Bind our shader program to our shader files (only need to run once per loop for both shaders)
    fragmentShader.bind();

    //Update our vertex shader data using our transform operations and our camera for projection
    vertexShader.update(transform, camera);

    //Actually draw the mesh to the screen
    mesh.draw();

    //Update our fragment shader data using our window resolution
    fragmentShader.update(glm::vec2((float)window.getWidth(), (float)window.getHeight()));

    //Swap buffers and input poll
    window.update(camera);

    //Increment our counter -- make smaller for fast processors, larger for slow processors
    transformCounter += 0.0006f;
}
```

Analyzing the Source Code

What Does Our Mesh Look Like?

```
//Hardcode a vertex array specifying two triangles offset by 0.5 units in the z direction
Vertex vertex[6] =
{
    Vertex(glm::vec3(0.5f, -0.5f, 0.0f)),
    Vertex(glm::vec3(0.0f, 0.5f, 0.0f)),
    Vertex(glm::vec3(-0.5f, -0.5f, 0.0f)),

    Vertex(glm::vec3(0.5f, -0.5f, 0.5f)),
    Vertex(glm::vec3(0.0f, 0.5f, 0.5f)),
    Vertex(glm::vec3(-0.5f, -0.5f, 0.5f))
};


//Instantiate mesh using vertex data above & the size of a given vertex
Mesh mesh = Mesh(vertex, sizeof(vertex)/sizeof(vertex[0]));
```

- Ideally, meshes should be read in from files, not hardcoded

Analyzing the Source Code

How Does Our Mesh Get Drawn to the Screen?

Happen Sequentially in Main Loop

```cpp
void Shader::update(Transform& transform, Camera& camera)
{
    //Combine view projection matrix with the model/world matrix
    glm::mat4 mvp = camera.getViewProjection() * transform.getWorld();

    //Send our transformation and projection data to our vertex shader program
    glUniformMatrix4fv(uniforms[TRANSFORM_RESOLUTION], 1, GL_FALSE, &mvp[0][0]);
}
```

Shader.cpp

Transform & Projection Matrix Uniform Variable

```cpp
void Mesh::draw()
{
    //Re-bind vertex array
    glBindVertexArray(getVertexArrayObject());

    //Draw array data to screen
    glDrawArrays(GL_TRIANGLES, 0, getNumVertices());

    //Unbind again
    glBindVertexArray(0);
}
```

Mesh.cpp

Calls

```glsl
#version 120

attribute vec3 position;

uniform mat4 transform;

void main()
{
    gl_Position = transform * vec4(position, 1.0);
}
```

VertexShader.vs

Analyzing the Source Code

How Does Our Mesh Get Colored?

```
#version 120

attribute vec3 position;

uniform mat4 transform;

void main()
{
    gl_Position = transform * vec4(position, 1.0);
}
```

VertexShader.vs

```
void Shader::update(glm::vec2 resolution)
{
    //Send our resolution data to our fragment shader program
    glUniform2fv(uniforms[TRANSFORM_RESOLUTION], 1, &resolution[0]);
}
```

Shader.cpp

Vertex Stream

Resolution Uniform Variable

Happen Sequentially in OpenGL Pipeline

```
#version 120

uniform vec2 resolution;

void main()
{
    vec2 st = gl_FragCoord.xy/resolution;
    gl_FragColor = vec4(0.25, st.y * gl_FragCoord.w, st.x * gl_FragCoord.w, 1.0);
}
```

FragmentShader.fs

The Result

What Does It Look Like?

Writing Camera Transformation Functions

Writing a Camera Rotation Function (1 of 4)

- **Goal**: Use mouse input to rotate the camera
  - Intuitively, this might be called “looking around”
- Important collaborators in this process
  - Mouse x-coordinate, Mouse y-coordinate, Window aspect ratio, Camera up-vector, Camera forward-vector, Camera view projection

Any Ideas?

Writing Camera Transformation Functions

Writing a Camera Rotation Function (2 of 4)

- Rotation left & right
  - Forward vector ([0, 0, 1] by default) x & z-coordinates “rotate” using trigonometric functions
  - Z-X plane where the z-axis is horizontal and the y-axis is vertical
  - Sine & Cosine imply some rotation
    - Sine returns the vertical coordinate, Cosine returns the horizontal coordinate
  - We will use the mouse’s normalized x-coordinate as input, multiplied by some radian angle
    - This isn’t necessarily the best design
    - It places a limit on how far upward and downward we can rotate
    - Better design would involve more persistence, and low-level control over our mouse

Writing Camera Transformation Functions

Writing a Camera Rotation Function (3 of 4)

- Rotation up & down
  - Up vector ([0, 1, 0] by default) y & z-coordinates “rotate” using trigonometric functions
  - Y-Z plane where the y-axis is horizontal and the z-axis is vertical
  - Sine & Cosine imply circular rotation
    - Sine returns the vertical coordinate, Cosine returns the horizontal coordinate
  - We will use the mouse’s normalized y-coordinate as input, multiplied by some radian angle
    - This isn’t necessarily the best design
    - It places a limit on how far upward and downward we can rotate
    - Better design might involve more persistence, and low-level control over our mouse

Writing Camera Transformation Functions

Writing a Camera Rotation Function (4 of 4)

- Rotation left & right (revisited)
  - Forward-vector & up vector MUST REMAIN PERPENDICULAR!
  - We need to update the forward-vector's y-coordinate using the up-vector
    - If the up-vector's y is 1 (as it is by default), then the forward-vector's y should be 0 (also as it is by default)
    - So, we can't use the up-vector's y-coordinate for this (without some extra, unneeded work)
  - How about the up-vector's z-coordinate?
    - The up-vector's z-coordinate increases as its y-coordinate decreases, vice versa
    - By default, the up-vector's z-coordinate is 0
    - Sounds pretty perfect to use to transform the forward vector
      - Forward vector's y-coordinate = (default value - up-vector's y-coordinate)

Writing Camera Transformation Functions

Writing a Camera Translation Function (1 of 3)

- **Goal**: Use keyboard input to translate the camera
  - Intuitively, this might be called “moving around”
- Important collaborators in this process
  - W-A-S-D Keys, Camera position vector, Camera up-vector, Camera forward-vector, Camera view projection

Any Ideas?

Writing Camera Transformation Functions

Writing a Camera Translation Function (2 of 3)

- Motion forward & backward
  - Position vector ([0, 0, -3] by default)
  - Perhaps we could just add/subtract the forward vector onto/from the position vector on key input?
    - Addition = forward
    - Subtraction = backward
  - In order to reduce choppiness, we should locally scale down the forward vector before adding/subtracting it onto/from the position vector

Writing Camera Transformation Functions

Writing a Camera Translation Function (3 of 3)

- Motion left & right
  - Position vector ([0, 0, -3] by default)
  - What is left & right relative to the camera?
    - Remember, we can rotate the camera around in space, and so left & right changes on this rotation
  - Well, left & right with regard to the direction that the camera is facing can be determined by taking the cross-product of the forward and up-vectors
    - This is why they need to stay perpendicular!
    - This returns a vector perpendicular to BOTH the forward & up-vectors
      - Under this context, it returns a “right-vector”
      - We can simply add a scaled “right-vector” to the camera’s position vector to move right
      - We can subtract a scaled “right-vector” from the camera’s position vector to move left

Works Cited

Thanks for stopping by!

- AtSpace (n.d.), Tutorial 12 - Perspective Projection. *AtSpace*, Accessed 12 January 2021, http://ogldev.atspace.org/www/tutorial12/tutorial12.html
- The Benny Box (9 January 2014), Intro to Modern OpenGL. *YouTube,* Accessed 2 January 2021, https://www.youtube.com/watch?v=ftiKrP3gW3k&list=PLEETnX-uPtBXT9T-hD0Bj31DSnwio-ywh
- Carnegie Mellon (n.d.), History of Computer Graphics (CG). *The Computer Graphics Book of Knowledge,* Accessed 8 January 2021, https://www.cs.cmu.edu/~ph/nyit/masson/history.htm
- De Vries, Joey (n.d.), Geometry Shader. *learnopengl*, Accessed 8 January 2021, https://learnopengl.com/Advanced-OpenGL/Geometry-Shader
- Gonzalez Vivo, Patricio & Lowe, Jen (n.d.), The Book of Shaders. *thebookofshaders*, Accessed 8 January 2021 https://thebookofshaders.com/03/