MATRICES CONTINUED…

GAME 300

James Dupuis

Objects

- Review
  - Matrix mode code and perspective setup

- Today:
  - Matrix Stacks
    - Pushing and popping
  - Matrix Calculations from transformations
  - Custom Matrices

COORDINATES

- In our previous discussions we analyzed the glTranslate, glRotate and glScale API calls of OpenGL.
  - We recall that we can reset our matrix to the identity by calling glLoadIdentity().
  - We can then apply modifications to the matrix which are incremental.
    - We append the latest transformations changes to the matrix.
- What if we had a world where we had rows of houses, and inside each house we had AI NPC’s confined?
  - The AI could walk around within the restrictions of the house it resides, but not outside.
  - The house contains things like a fridge which can store items inside of it.

GLOBAL COORDINATES

- We would need to keep track of the offset in x and z of each house.
- We would then need to keep track of the AI within the houses x, y and z
- We would also need to keep track of items placed within the house, and perhaps even items placed within items within the house.
  - We could theoretically keep track of everything in one large coordinate system where X, Y and Z are a very large scale.
  - This is known as **Global Coordinates**.
  - Using only Global Coordinates makes things difficult to manage and write functions for.
- Say we wanted to write a function which limits the NPC AI to the constraints of the house?
  - We would need different coordinates to check for the bounds of each house.

LOCAL COORDINATES

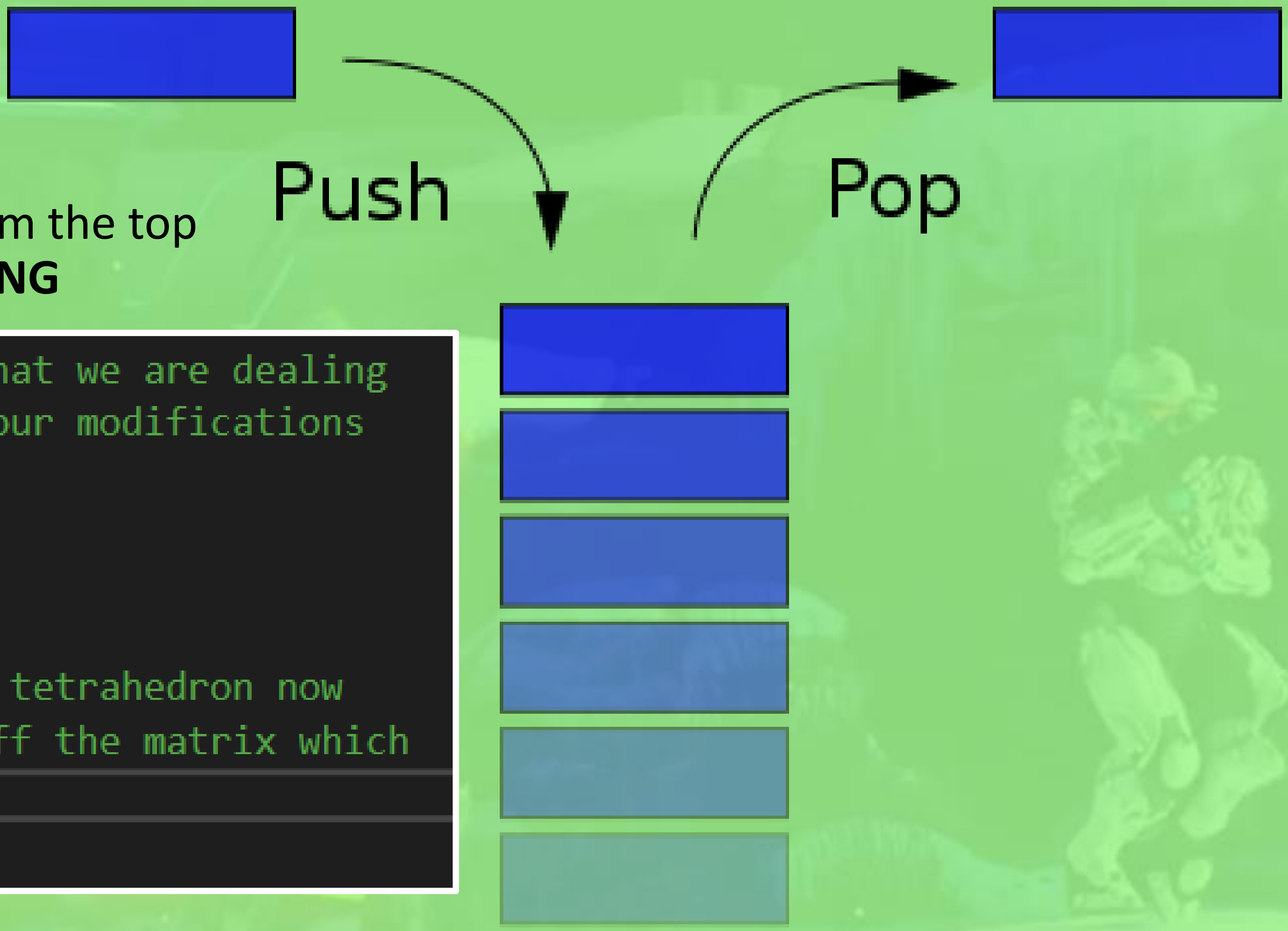
- We can instead solve this problem by keeping track of coordinates locally.
  - Nested Objects need to only manage local changes in coordinates.
- When we process the changes, we simply take any parent coordinate systems and append them to the local coordinates.
  - Example:
    - House #2 (Global Coordinate within map)
    - (200, 10, 50)
      - AI (local coordinate within the house)
      - (5, 3, 12)
      - Fridge (local Coordinate within the house)
      - (6, 3, 12)
        - Ice cube tray, local coordinate within fridges local coordinate)
        - (1.0, -1.0, 1.5),
          - Ice cube (local coordinate within ice cube trays local coordinate system)
          - (0.5, 0, 0.5)

MATRIX STACKS

- Ideally we can keep track of each coordinate system separately so that coordinates are manageable.
  - OpenGL has what is known as Matrix stacks to help manage this.
- Both the projection Matrix and the ModelView Matrix has a series of layers known as the matrix stack.
  - The Modelview matrix has the ability to track at least 32 layers worth of matrices.
  - The Projection Matrix has only 2 layers.
- The way the data is managed like a stack of papers piled on top of each other.
  - You can take a piece from the bottom or middle
    - (that might make our stack of papers tip over)
  - We instead can only take off the top or put more on top of the pile.

PUSHING & POPPING

- The putting a new matrix on top of the stack is known at **PUSHING**
- The act of taking off a layer from the top of the stack is known as **POPPING**




```
// we push the matrix here so that we are dealing
// with a clear matrix for all our modifications
// of this object to remain on.
glPushMatrix();
myObj.Update();
myObj.Draw();
// we are done dealing with the tetrahedron now
// so we can go ahead and pop off the matrix which
// contained it's modifications
glPopMatrix();
```

TREE EXAMPLE

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();

// this code would be nested in objects but is in one file to
// keep it simplified as an example...

// tree transformation section
glPushMatrix();
    // place tree on map
    myTree.Update();
    myTree.Draw();
    for (int i = 0; i < NUM_BRANCHES_PER_TREE; i++)
    {
        // keeps track of only modifications to the individual
        // branch on the tree
        glPushMatrix();
            // rotate and translate randomly per branch
            branches[i].Update();
            branches[i].Draw();
        glPopMatrix();
    }
glPopMatrix();
```

Matrices

- Matrices in Graphics Programming are used to store a series of transformation changes to things like:
  - Objects within the world
  - The world itself
  - The camera system

- A matrix, as you are learning in your math, is a series of numbers stored in rows and columns.
  - In programming it would seem obvious to code this simply as a 2D Array.
    - In OpenGL, however, we use a single array the size of the entire elements of the matrix.
    - So a 4x4 matrix would become a float[16].

Matrices (Column vs Row Order)

- OpenGL and DirectX store values inside of matrices in a slightly different fashion.
- Both can store Matrices as single array of size 16, however, both have a standardized difference in terms of how that array is organized.

  - row major
  - [0, 1, 2, 3,
    4, 5, 6, 7,
    8, 9, 10, 11,
    12, 13, 14, 15]

  OR

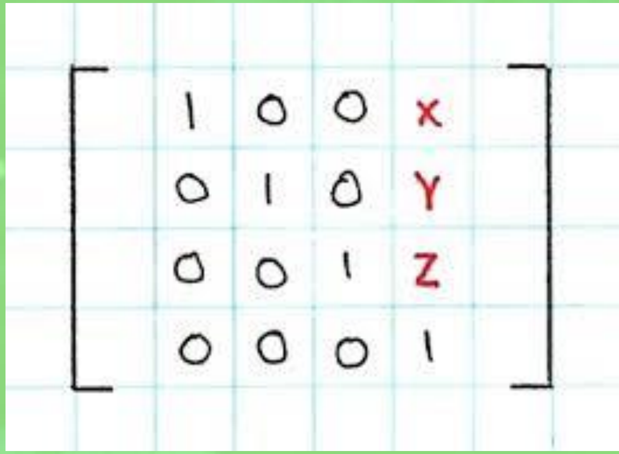
  column major

  [0, 4, 8, 12,
  1, 5, 9, 13,
  2, 6, 10, 14,
  3, 7, 11, 15]

- It's important to understand the differences as values in the wrong position could cause drastically different results.
- OpenGL is typically Column Ordered and DirectX is row ordered.
  - This means that OpenGL counts column by column top to bottom for it's values
  - DirectX goes row to row, left to right.
    - *OpenGL can support row order but most libraries assume column order is being used.

Translation Matrix Values

- The translation Matrix starts out similar to the Identity Matrix with 1's along the same diagonal line.
  - The part of the matrix where the translation occurs is the last column:
- The X, Y, & Z value is the 13thth -> 15th matrix values.
  - The reason we use vec4 for positions vs vec3's becomes more apparent when we reflect on the use of this matrix multiplication.
    - If we were to multiply a vec4 of (1,2,3, 0) vs a vec4 of (1,2,3,1) we would get vastly different results.
    - Positions use 1 as it's final value to indicate it can be translated where as vectors should not be translated and applying a 0 allows them to remain in tact.



Why Vec4's?

- vec4's allow us, the programmer, to tag vec3's with an additional value.
  - This allows us to differentiate between whether a vector is representing a direction or a magnitude (length).
  - Called homogeneous vectors.
    - W = 0 direction
    - W = 1 Coordinate
  - The W axis can also be used for scaling with projection calculations.
- Also allows us to apply our vectors to a 4x4 Matrix for transformations.
  - Vec3's cannot be multiplied or applied directly to a 4x4 Matrix.
  - Everything within graphics typically uses 4x4 Matrices. (more later)
- We can determine other variables for 3D calculations given 2 or more vectors:
  - Dot Product
  - Cross Product
  - Length of a Vector

Vmath

- To make our lived easier with dealing with 3D math, there is a library called vmath.
  - Vmath saves time recreating the wheel.
  - Gives us an idea of the modifications that our translate, scale, and rotate calls are making behind the scenes.
- Used in the C++ side of your openGL applications, we will use this library when we start working with shader code.
  - Contains a vec3 class as well as a vec4 class.
    - vmath::vec3 myVec3(1.0f, 1.0f, 1.0f);
    - vmath::vec4 myVec4(1.0f, 1.0f, 1.0f, 1.0f);
- If you need to convert a vec3 to a vec4 you can either copy each individual index element or you can copy using the following:
  - vec3 myVec1(1.0f, 1.0f, 1.0f);
  - vec4 myVec2(myVec1, myVec2);
- Contains operator handling to allow Vectors and Matrices to be added subtracted, multiplied, etc…

Translation Matrix Values

- Vmath has a set of functions to handle translations of a matrix for you so you don't have to worry about the matrix assignments.
  - Translate(x,y,z) or translate(vec4());

```
// This code moves the cube to the correct position
vmath::mat4 mv_matrix = vmath::translate(-2.25f + (x * CUBE_WIDTH), -2.0f + (y * CUBE_HEIGHT), -5.0f + (z * -CUBE_LENGTH));
```

```
template <typename T>
static inline Tmat4<T> translate(T x, T y, T z)
{
    return Tmat4<T>(Tvec4<T>(1.0f, 0.0f, 0.0f, 0.0f),
                    Tvec4<T>(0.0f, 1.0f, 0.0f, 0.0f),
                    Tvec4<T>(0.0f, 0.0f, 1.0f, 0.0f),
                    Tvec4<T>(x, y, z, 1.0f));
}
```

$$\begin{bmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & Y \\ 0 & 0 & 1 & Z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Scale Matrix Values

- The scaling section is probably the simplest of transformation matrices to understand outside of the identity.
  - That’s because the scaling matrix pretty much is the identity Matrix with one major exception.
  - Instead of 1’s diagonally is takes the x,y, & z values diagonally to the amount to scale in each direction.

$$\begin{bmatrix} s_x & 0.0 & 0.0 & 0.0 \\ 0.0 & s_y & 0.0 & 0.0 \\ 0.0 & 0.0 & s_z & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 \end{bmatrix}$$

- *note: when an object scales. It scales in both negative and positive values of that axis.

Scale Matrix Values

- Again, Vmath has you covered with the scale function:

$$\begin{bmatrix} s_x & 0.0 & 0.0 & 0.0 \\ 0.0 & s_y & 0.0 & 0.0 \\ 0.0 & 0.0 & s_z & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 \end{bmatrix}$$

```
template <typename T>
static inline Tmat4<T> scale(T x, T y, T z)
{
    return Tmat4<T>(Tvec4<T>(x, 0.0f, 0.0f, 0.0f),
                    Tvec4<T>(0.0f, y, 0.0f, 0.0f),
                    Tvec4<T>(0.0f, 0.0f, z, 0.0f),
                    Tvec4<T>(0.0f, 0.0f, 0.0f, 1.0f));
}
```

Rotation Matrix Values

- The last matrix for transformations is the rotation matrices…
- Rotations occur "about" an axis or revolve around an axis.
- Because of this the rotation matrix isn't very straightforward compared to the translation and matrices.
  - Each axis rotation has it's own matrix.
  - To further complicate things, rotation matrices require the use of cos and sin calculations on the angle to rotate about the axis.

$$R_x = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$R_y = \begin{pmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$R_z = \begin{pmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Composite Rotation Matrix

- A composite transform Matrix of the 3 rotational matrices combined looks something like:

$$R_z(\psi)\,R_y(\theta)\,R_x(\phi) = \begin{bmatrix} c_\theta c_\psi & c_\phi s_\psi + s_\phi s_\theta c_\psi & s_\phi s_\psi - c_\phi s_\theta c_\psi & 0.0 \\ -c_\theta s_\psi & c_\phi c_\psi - s_\phi s_\theta s_\psi & s_\phi c_\psi + c_\phi s_\theta s_\psi & 0.0 \\ s_\theta & -s_\phi c_\theta & c_\phi c_\theta & 0.0 \\ 0.0 & 0.0 & 0.0 & 1.0 \end{bmatrix}$$

- Vmath of course has a function available to make your life easier:
  - rotate(angle, x,y,z) which will rotate the amount of angle (degrees) defined around the specified axis where x,y,z are normalized values (0->1)
  - Shown on next slide…

Vmath Rotate Function

```cpp
template <typename T>
static inline Tmat4<T> rotate(T angle, T x, T y, T z)
{
    Tmat4<T> result;

    const T x2 = x * x;
    const T y2 = y * y;
    const T z2 = z * z;
    float rads = float(angle) * 0.0174532925f;
    const float c = cosf(rads);
    const float s = sinf(rads);
    const float omc = 1.0f - c;

    result[0] = Tvec4<T>(T(x2 * omc + c), T(y * x * omc + z * s), T(x * z * omc - y * s), T(0));
    result[1] = Tvec4<T>(T(x * y * omc - z * s), T(y2 * omc + c), T(y * z * omc + x * s), T(0));
    result[2] = Tvec4<T>(T(x * z * omc + y * s), T(y * z * omc - x * s), T(z2 * omc + c), T(0));
    result[3] = Tvec4<T>(T(0), T(0), T(0), T(1));

    return result;
}
```

$$R_x = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$R_y = \begin{pmatrix} \cos\theta & 0 & \sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

$$R_z = \begin{pmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

PRINTING OUT THE MATRIX VALUES

- Our Matrices are stored in OpenGL as an array of float[16].
- We can retrieve the values of the current matrix using the glGetFloatv function like so:

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
printMVMatrix();

glPushMatrix();
    // modify object transformation
    myObj.Update();
    myObj.Draw();
    printMVMatrix();
glPopMatrix();

SDL_GL_SwapWindow(window);
```

```
void printMVMatrix()
{
    GLfloat currMatrix[16];
    glGetFloatv(GL_MODELVIEW_MATRIX, currMatrix);
    for (int i = 0; i < 4; i++)
    {
        cout << currMatrix[i] << '\t';
        cout << currMatrix[i+4] << '\t';
        cout << currMatrix[i+8] << '\t';
        cout << currMatrix[i+12] << '\t';
        cout << endl;
    }
    cout << endl;
}
```

```
D:\FALL 2019\GAME 300 - Graph
1       0       0       0
0       1       0       0
0       0       1       0
0       0       0       1


1       0       0       0
0       1       0       -1
0       0       1       0
0       0       0       1
```

- The Example shows a matrix with a single translation applied in the y axis.

Summary

- Learned about:
  - Global vs local coordinate systems
  - Matrix Stacks
    - Pushing & Popping
  - Vmath library
  - How transformations modify the matrices
- After the break:
  - Collision detection