CS 4458 Lecture 09

- Outcomes

  - Understand and be able to use transformations in 3-space
  - Understand and be able to use frustum culling
  - Understand the use of space Partitioning via Quad and Oct Trees
  - Understand and be able to use culling queries for occlusion
  - Understand and be able to use Timer Queries for performance measurement
  - Understand the use of Euler Angles and their issues including Gimbal Lock
  - Understand and be able to use Quaternions for rotations

• 3D Transformations Using Homogenous Coordinates

- The efficiency gains we saw for 2D transformations using Homogenous coordinates over cartesian coordinates are much higher when we go to 3D, and so there is no need to consider cartesian coordinates any further.

- So from now on we will use 3D homogenous coordinates, which in the general case will have the form: (x, y, z, w), but for now will be $(x, y, z, 1)$, as in the 2D case.

- The point in vector form will be:

$$P = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

- Translation and scaling matrices follow directly from the 2D analysis, and are

Translation: $$P' = \begin{bmatrix} 1 & 0 & 0 & dx \\ 0 & 1 & 0 & dy \\ 0 & 0 & 1 & dz \\ 0 & 0 & 0 & 1 \end{bmatrix} P$$

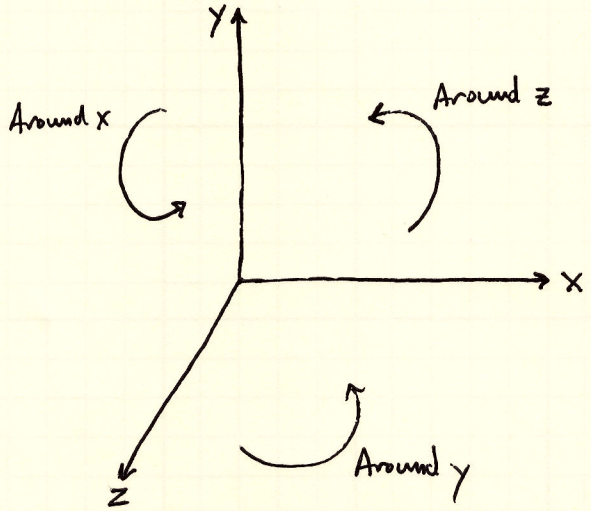
Scaling: $$P' = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} P$$

- Rotation is more complicated, because it is not immediately clear what we should be rotating around in a 3D environment. We could rotate around an arbitrary axis, but most users have a hard time visualizing what is going to happen if, for example, they rotate by 30° around an axis with direction vector $(2, 3, -1)$

- To avoid this problem we will only ~~need~~ consider rotations around one of the three primary axes, x, y, z. Any complex rotation will have to be built up as a sequence of rotations around these axes.

- The direction of rotation will, by generally accepted convention, be CCW when looking down the axis toward the origin

- As the picture below shows, a rotation around the x-axis will move parallel to the y/z plane in the direction from y to z, a rotation around the y-axis will move parallel to the z/x plane in the direction from z to x and a rotation around the z-axis will move parallel to the x/y plane in the direction from x to y:



- Prior to getting into matrices, we should note one potential problem, which is that if one does rotations by breaking them down into rotations around the primary axes, then it is not commutative. That is, the order of rotations matters.

- The simplest rotation matrix to develop is the rotation around z, which we will call R_z. When we do this the effect on x and y values will mirror the 2D case, and the z value will be unchanged, and so the formula will be:

$$P' = \begin{bmatrix} \cos(\theta) & -\sin(\theta) & 0 & 0 \\ \sin(\theta) & \cos(\theta) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} P$$

- We can now derive R_x and R_y from R_z by symmetry:

$$R_x(\theta) = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\theta) & -\sin(\theta) & 0 \\ 0 & \sin(\theta) & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R_y(\theta) = \begin{bmatrix} \cos(\theta) & 0 & \sin(\theta) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

- The only thing to be careful of here is the location of the negative sign on the sin function in R_y. In R_z the rotation is from x to y, and the negative sign is on the xy element (row 1, column 2) of the matrix. So since R_y rotates from z to x, symmetry requires that the negative sign must be on the zx element of the matrix (row 3, col 1)

- Example: Build the combined transformation matrix C, for a clockwise rotation of 90° around the y-axis fixing the point (1,0,1) followed by a scale of (2,1,1) fixing the point (0,1,1). Apply the matrix to the Cartesian point (1,2,-1)

The order of the operations will be:

1. $T(-1,0,-1)$
2. $R_y(-90°)$ // remember default is CCW
3. $T(1,0,1)$
4. $T(0,-1,-1)$
5. $S(2,1,1)$
6. $T(0,1,1)$

Remembering to put these right to left, and noting that the CW rotation changes the signs of the sins, we have:

$$C = \begin{bmatrix}1&0&0&0\\0&1&0&1\\0&0&1&1\\0&0&0&1\end{bmatrix}\begin{bmatrix}2&0&0&0\\0&1&0&0\\0&0&1&0\\0&0&0&1\end{bmatrix}\begin{bmatrix}1&0&0&0\\0&1&0&-1\\0&0&1&-1\\0&0&0&1\end{bmatrix}\begin{bmatrix}1&0&0&1\\0&1&0&0\\0&0&1&1\\0&0&0&1\end{bmatrix}\begin{bmatrix}0&0&-1&0\\0&1&0&0\\1&0&0&0\\0&0&0&1\end{bmatrix}\begin{bmatrix}1&0&0&-1\\0&1&0&0\\0&0&1&-1\\0&0&0&1\end{bmatrix}$$

Calling these matrices A, B, C, D, E, F, and G, and taking these a pair at a time we have:

$$AB = \begin{bmatrix}2&0&0&2\\0&1&0&0\\0&0&1&1\\0&0&0&1\end{bmatrix} \quad DE = \begin{bmatrix}1&0&0&1\\0&1&0&-1\\0&0&1&0\\0&0&0&1\end{bmatrix} \quad FG = \begin{bmatrix}0&0&-1&1\\0&1&0&0\\1&0&0&-1\\0&0&0&-1\end{bmatrix}$$

$$ABDE = \begin{bmatrix}2&0&0&2\\0&1&0&0\\0&0&1&1\\0&0&0&1\end{bmatrix} \quad C = ABDEFG = \begin{bmatrix}0&0&-2&4\\0&1&0&0\\1&0&0&0\\0&0&0&1\end{bmatrix}$$

Applying this matrix to the Cartesian Point (1,2,-1):

$$\begin{bmatrix}0&0&-2&4\\0&1&0&0\\1&0&0&0\\0&0&0&1\end{bmatrix}\begin{bmatrix}1\\2\\-1\\1\end{bmatrix} = \begin{bmatrix}6\\2\\1\\1\end{bmatrix}$$

And so (1,2,-1) is transformed to (6,2,1)

Frustum Culling and Space Partitioning

- **Problem:** In a large scene with a significant amount of polygons the rendering will be complex and time consuming, furthermore as the scene grows or as animation is put in place the program will slow down.

  This is complicated by the notion that the clipping planes are suppose to remove those polygons which will not be visible, yet the slow down still occurs.

  Why is this? Basically because the removal happens late in the pipeline after transformation occurs

  What we need is a method to prevent these polygons from entering the pipeline in the first place.

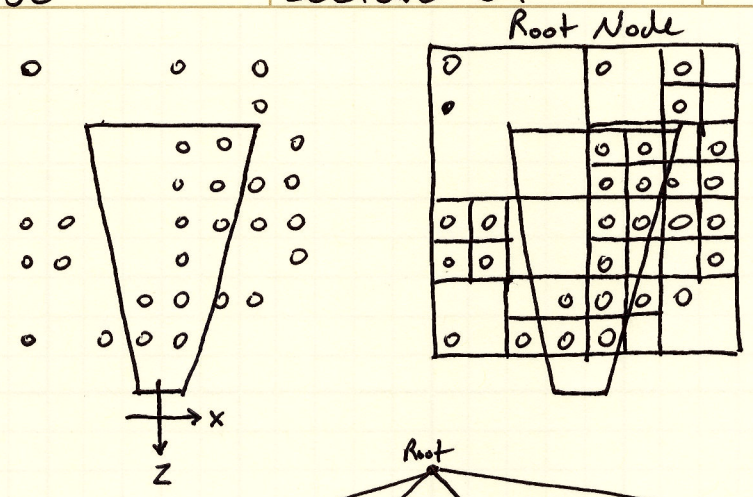
  Essentially, we want to remove any object that does not appear in the frustum. This process is called frustum culling.

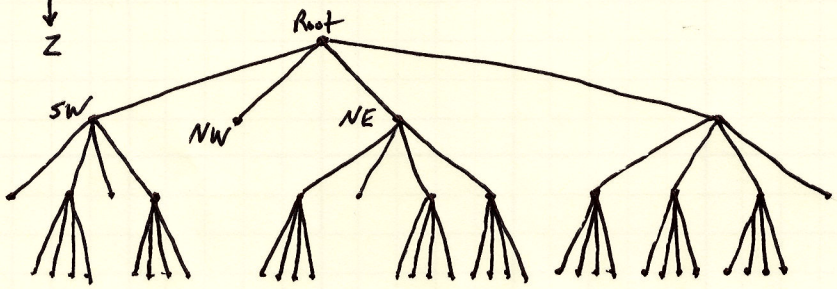
- Space Partitioning

  - The naive approach to handling frustum culling would be to up front check each object's intersection with the frustum.
  - As this tends to be costly when there are a lot of objects or objects which are very complex, we need a more efficient approach. One such approach is Space Partitioning
  - Here the distribution of the objects drives the culling. We divide the space up into a set ~~of~~ of cells, each of which may contain many objects, and then check if the cell intersects the frustum
    - Distribution check and partitioning is a one-time up front cost (unless objects are not static
    - Cell shape allows for easy intersection checks
    - Hierarchal nature of partitioning allows for pruning
    - Cells are subdivided into a partition representing object distribution, using an efficient process

- Quad Trees

  - Start with initial node representing entire space
  - At each level a node can be subdivided into 4 equal sub-nodes (quadrants)
    - Typically when a node intersects multiple objects
  - When subdivided each quadrant becomes a child of the subdivided node
  - Culling starts at the root, and for every non-leaf node which intersects the frustum its children are recursively processed.





- OctTrees and BSP Trees
  - Generalization of QuadTrees to 3-space, but rather than partitioning to squares, octtrees partion to a hierarchy of axis-aligned cubes
  - Additionally one can also use Binary Space Partitioning Trees
  - These techniques extend beyond Frustum culling and into collision detection and other problems
  - One issue is that of mobile objects and redistribution handling, which can increase the overhead of using such techniques.

◦ Occlusion

- Often from a global view of a scene we can determine that an object is occluded, or blocked, from view, and thus need not be sent to the pipeline for processing
- In such cases we can apply occlusion culling to improve rendering efficiency. Occlusion culling amounts to effectively applying an occlusion query to determine if an object is visible after depth testing
  - This is done by surrounding the drawing commands with glBeginQuery(...) and glEndQuery
  - The process is as follows:
    1.) In setup, call glGenQueries(num, *idvar) which will generate the number of query objects and store their ids in idvar
    2.) In the drawing function call

```
glBeginQuery(GL_SAMPLES_PASSED, idvar);
// drawing code
glEndQuery(GL_SAMPLES_PASSED);
```

counts the number of fragments which pass the depth test

    3.) Ensure the results are available

```
while (!resultAvailable)
    glGetQueryObjectuiv(idvar, GL_QUERY_RESULT_AVAILABLE,
        &resultAvailable);
```

    4.) Collect results

```
glGetQueryObjectuiv(idvar, GL_QUERY_RESULT, &result)
```

    5.) Conditional Processing → Culling

```
if (result) {
    // conditional drawing code
}
```

- Conditional Rendering
  - As of OpenGL 3.0, we can apply conditional rendering which is a far more efficient method of handling the query outcome.
  - The programmer can specify a block of drawing command which will only be executed if a query comes back positive
  - This is done as follows:

```
glBeginConditionalRender(query, GL_QUERY_WAIT);
// conditional drawing commands
glEndConditionalRender()
```

→ wait for query to complete
→ id of query to execute

Timer Queries and Performance Measurement

- In prior lectures we noted that we can use frame rates to evaluate program performance, but this is a very blunt instrument. It does not provide a precise enough measurement to inform us which code is slowing down a program.
- If we are interested in truly fine tuning our program we will need another approach. Luckily queries can help us here as well.
- If we suspect a block of code is slowing down our program we can place the following query around it

```
glBeginQuery(GL_TIME_ELAPSED, query[0]);
// Drawing code -> suspicious
glEndQuery(GL_TIME_ELAPSED);
```

  - We can then retrieve the value as follows

```
glGetQueryObjectui64v(query[0], GL_QUERY_RESULT, ----)
```

  - Note: The value is in nanoseconds

Quaternions and Rotations

• Definition and Notation

- First described by Hamilton in 1844 and 1845, and are essentially multi-dimensional complex numbers
- Whereas standard complex numbers have a scalar component and an imaginary component, with quaternions the imaginary part is an imaginary vector based on three imaginary orthogonal ones.
- A quaternion is defined using a scalar component, w, and an imaginary component $\vec{v} = \langle iv_1, jv_2, kv_3 \rangle$. Those can be written in three equivalent ways:
  - $q = (w, \vec{v})$
  - $q = (w, v_1, v_2, v_3)$
  - $q = (w + iv_1 + jv_2 + kv_3)$
- If the scalar component, w, is zero, then the quaternion is called a pure quaternion.
- First, we need to define i, j, k. They are imaginary orthogonal unit vectors that satisfy the following equations:
  - $i^2 = j^2 = k^2 = -1$
  - $ij = k = -ji$
  - $jk = i = -kj$
  - $ki = j = -ik$
- We will also need a couple of quaternion properties, the magnitude of a quaternion and how to multiply two quaternions.
  - The size, or magnitude, of $q = (w, \vec{v})$ is given by

$$|q| = \sqrt{w^2 + |\vec{v}|^2}$$

  - Quaternion multiplication is defined in the obvious way:

$$q_1 q_2 = (w_1 + v_{11}i + v_{12}j + v_{13}k)(w_2 + v_{21}i + v_{22}j + v_{23}k)$$

$$= (w_1 w_2 + w_1 v_{21} i + w_1 v_{22} j + w_1 v_{23} k + v_{11} w_2 i + \ldots + v_{13} v_{23} k^2)$$

  - Then we just have to substitute k for ij, -1 for i^2, etc., in these 16 terms and we are done
  - We can also, by rearranging the terms, define quaternion multiplication using dot product and cross product to simplify the result

- The scalar terms in the product above will come from

$$\begin{aligned} w &= w_1 w_2 + v_{11} v_{21} i^2 + v_{12} v_{22} j^2 + v_{13} v_{23} k^2 \\ &= w_1 w_2 - (v_{11} v_{21} + v_{12} v_{22} + v_{13} v_{23}) \\ &= w_1 w_2 - \vec{v}_1 \cdot \vec{v}_2 \end{aligned}$$

  - First replacing the square terms with -1 and then using the definition of the dot product

- We can collect the other terms, and get:

$$q_1 q_2 = (w_1 w_2 - \vec{v}_1 \cdot \vec{v}_2, w_1 \vec{v}_2 + w_2 \vec{v}_1 + \vec{v}_1 \times \vec{v}_2)$$

- Using Quaternions for Rotation Around an Arbitrary Axis

  - To rotate a point (x, y, z) by an angle θ counterclockwise around an arbitrary axis through the origin which is defined by a unit vector $\vec{u}$, we compute

$$P' = q P q^{-1}$$

  - where:

    $P = (0, x, y, z)$ is a pure quaternion whose imaginary part is the point

$$q = \left(\cos\left(\frac{\theta}{2}\right), \sin\left(\frac{\theta}{2}\right)\vec{u}\right)$$

$$q^{-1} = \left(\cos\left(\frac{\theta}{2}\right), -\sin\left(\frac{\theta}{2}\right)\vec{u}\right)$$

  - The three quaternions on the right hand side of the equation for P' are multiplied using quaternion multiplication

  - The new quaternion P' will, like P, be pure (i.e., its real component will be zero), and its imaginary components will contain the transformed part.

• Quaternion Rotation Example

Assume that we want to rotate the point $(1,2,-1)$ 90° around the z-axis, which is defined by the unit vector $(0,0,1)$. Since z will be fixed, this is a simple 2D rotation, and we can easily see that the new point should be $(-2,1,1)$.

Since $\sin(45°) = \cos(45°) = \frac{1}{\sqrt{2}}$, we have

$$q = \left(\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}}(0,0,1)\right) = \left(\frac{1}{\sqrt{2}}, 0, 0, \frac{1}{\sqrt{2}}\right)$$

$$p = (0,1,2,-1)$$

$$q^{-1} = \left(\frac{1}{\sqrt{2}}, -\frac{1}{\sqrt{2}}(0,0,1)\right) = \left(\frac{1}{\sqrt{2}}, 0, 0, -\frac{1}{\sqrt{2}}\right)$$

and so,

$$qp = \left(\frac{1}{\sqrt{2}} + \frac{1}{\sqrt{2}}k\right)(i + 2j - k)$$
$$= \frac{1}{\sqrt{2}}(1+k)(i+2j-k)$$
$$= \frac{1}{\sqrt{2}}(i + 2j - k + ki + 2kj - k^2)$$
$$= \frac{1}{\sqrt{2}}(i + 2j - k + j - 2i + 1)$$
$$= \frac{1}{\sqrt{2}}(1 - i + 3j - k)$$

$$qpq^{-1} = \frac{1}{\sqrt{2}}(1 - i + 3j - k)\frac{1}{\sqrt{2}}(1 - k)$$
$$= \tfrac{1}{2}(1 - i + 3j - k + ik - 3jk - k + k^2)$$
$$= \tfrac{1}{2}(1 - k - i - j + 3j - 3i - k - 1)$$
$$= \tfrac{1}{2}(0 - 4i + 2j - 2k)$$
$$= (0, -2, 1, -1)$$

- p', the transformed point, is the imaginary part of this pure quaternion which is $(-2,1,-1)$, as expected.
- As an alternative to the individual multiplications we could have (twice) used the definition of quaternion multiplication that uses dot and cross products given previously

$$q_1 q_2 = (w_1 w_2 - \vec{v}_1 \cdot \vec{v}_2, w_1 \vec{v}_2 + w_2 \vec{v}_1 + \vec{v}_1 \times \vec{v}_2)$$

- Using this approach we would have had

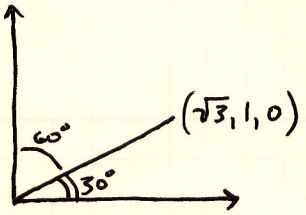
$$qpq^{-1} = \tfrac{1}{2}(1,(0,0,1))(0,(1,2,-1))(1,(0,0,-1))$$
$$= \tfrac{1}{2}(0 - (-1), (1,2,-1) + (-2,1,0))(1,(0,0,-1))$$
$$= \tfrac{1}{2}(1,(-1,3,-1))(1,(0,0,-1))$$
$$= \tfrac{1}{2}(1 - 1, (0,0,-1) + (-1,3,-1) + (-3,-1,0))$$
$$= \tfrac{1}{2}(0,(-4,2,-2))$$

- which is once again the point $(-2,1,-1)$

- Another Rotation Example
  - Now we'll try rotating the point $(\sqrt{3}, 1, 0)$ counterclockwise 60° around the z-axis
  - Since $\cos(30°) = \frac{\sqrt{3}}{2}$ and $\sin(30°) = \frac{1}{2}$, we have:

$$qP'q^{-1} = \left(\tfrac{\sqrt{3}}{2}, 0, 0, \tfrac{1}{2}\right)(0, \sqrt{3}, 1, 0)\left(\sqrt{3}/2, 0, 0, -\tfrac{1}{2}\right)$$
$$= \tfrac{1}{4}(\sqrt{3} + k)(\sqrt{3}\,i + j)(\sqrt{3} - k)$$
$$= \tfrac{1}{4}(3i + \sqrt{3}j + \sqrt{3}ki + kj)(\sqrt{3} - k)$$
$$= \tfrac{1}{4}(3i + \sqrt{3}j + \sqrt{3}j - i)(\sqrt{3} - k)$$
$$= \tfrac{1}{2}(i + \sqrt{3}j)(\sqrt{3} - k)$$
$$= \tfrac{1}{2}(\sqrt{3}i - ik + 3j - \sqrt{3}jk)$$
$$= \tfrac{1}{2}(\sqrt{3}i + j + 3j - \sqrt{3}i)$$
$$= 2j$$
$$= (0, 0, 2, 0)$$

  - So the transformed point is $(0, 2, 0)$. A simple diagram should persuade you that this is correct



- Multiple Rotations
  - Usually we won't be doing a single rotation, but will be doing a series of rotations in an animation. E.g., say that we have a sequence of rotations defined, in time order, by the quaternions $q_1, q_2, \ldots, q_n$. We will need to compute

$$P' = q_n \cdots q_2 q_1 P q_1^{-1} q_2^{-1} \cdots q_n^{-1}$$

  - Fortunately multiplication is associative, and also $q_1^{-1} q_2^{-1} \cdots q_n^{-1} = (q_1 q_2 \cdots q_n)^{-1}$ so we just use repeated multiplication to compute a single combined unit quaternion

$$q = q_n \cdots q_2 q_1 = (w, \vec{v})$$

and the inverse will be $(w, -\vec{v})$

CS 4458 Lecture 09

- For Next Time
  - Read Chapters 7, 8, 9
  - Review Prior Lectures
  - Continue working on HW 02
  - Come to Class

- Additional Notes