Geometric Objects & Transformations
Linear Algebra - Vectors

- Vectors operations
  - Addition
    \[ \mathbf{v}_1 + \mathbf{v}_2 = (a_1 + a_2, b_1 + b_2, c_1 + c_2) \]
  - Scalar multiplication
    \[ \lambda \mathbf{v} = (\lambda a, \lambda b, \lambda c) \]

- Norm
  \[ \| \mathbf{v} \| = \sqrt{a^2 + b^2 + c^2} \]
Linear Algebra - Vectors

- **Scalar Product**
  \[ v_1 \cdot v_2 = a_1 a_2 + b_1 b_2 + c_1 c_2 \]
  - Commutative
    \[ v_1 \cdot v_2 = v_2 \cdot v_1 \]
  - Linear
    \[ (\lambda v_1 + v_2) \cdot u = \lambda v_1 \cdot u + v_2 \cdot u \]
  - Orthogonally
    \[ v \perp u \iff v \cdot u = 0 \]
  - Orthonormality
    \[ ||v|| = ||u|| = 1 \]
  - Norm
    - Normalized vector
      \[ ||v|| = \sqrt{v \cdot v} \]
  - Vector
    \[ v = (a, b, c) \]
Cross products

- The cross product of two vectors \( \mathbf{v}_1 \) and \( \mathbf{v}_2 \) is defined (in determinant form)

\[
\mathbf{v}_1 \times \mathbf{v}_2 = \begin{vmatrix}
\hat{x} & \hat{y} & \hat{z} \\
x_1 & y_1 & z_1 \\
x_2 & y_2 & z_2
\end{vmatrix}
\]

\[
\|\mathbf{v}_1 \times \mathbf{v}_2\| = \|\mathbf{v}_1\|\|\mathbf{v}_2\|\sin(\theta)
\]

- Meanings
  - The area of the parallelogram that is defined by \( \mathbf{v}_1 \) and \( \mathbf{v}_2 \)
  - The direction of the normal to the plane that is determined by \( \mathbf{v}_1 \) and \( \mathbf{v}_2 \)
  - A vector which is orthogonal to both \( \mathbf{v}_1 \) and \( \mathbf{v}_2 \)
Cross Product

Properties

- \( \mathbf{v}_1 \times \mathbf{v}_2 = -\mathbf{v}_2 \times \mathbf{v}_1 \)
  - Reverse the direction.

- \( \| \mathbf{v}_1 \times \mathbf{v}_2 \| = \| \mathbf{v}_1 \| \| \mathbf{v}_2 \| \sin(\theta) \)
Vector Class

- C++ Class for handling vectors and vector operations

```cpp
class vec3 {
public:
    GLfloat x;
    GLfloat y;
    GLfloat z;

    vec3( GLfloat s = GLfloat(0.0) ) : 
        x(s), y(s), z(s) {}

    vec3( GLfloat x, GLfloat y, GLfloat z ) : 
        x(x), y(y), z(z) {}

    ...  
};
```
Vector Class

- Operator overloading

```cpp
vec3 operator + ( const vec3& v ) const {
    return vec3( x + v.x, y + v.y, z + v.z );
}

vec3 operator * ( const GLfloat s ) const {
    return vec3( s*x, s*y, s*z );
}

vec3 operator * ( const vec3& v ) const {
    return vec3( x*v.x, y*v.y, z*v.z );
}
...
```
Some Linear Algebra - Matrices

- Linear Transformation - Matrix Multiplication
  - Do not Commute
  \[
  (A^{n \times l} B^{l \times m})_{ij} = \sum_{k=1}^{l} A_{ik} B_{kj}
  \]
  \[
  AB \neq BA
  \]

- Matrix Transpose
  \[
  (A^T)_{ij} = A_{ji}
  \]
  \[
  (AB)^T = B^T A^T
  \]

- Matrix Inverse
  \[
  A^{-1}A = AA^{-1} = I
  \]
  \[
  (AB)^{-1} = B^{-1}A^{-1}
  \]
Determinants

- If $A$ is a $2 \times 2$ matrix then $\det(a) = ad - bc$
  - $\det(A) =$ area of parallelogram defined by $(a, c)$ and $(b, d)$

- General determinate is defined recursively by minors
  - Minors are computed by eliminating the row and column of an entry and computing the determinant on the residue.

$$|A_{n \times n}| = \sum_{j=1}^{n} a_{ij} (-1)^{i+j} M_{ij}$$

- Few reminders
  - $|AB| = |A||B|$
  - $|A^T| = |A|$
  - $|A^{-1}| = |A|^{-1}$

Basic Geometry - Center for Graphics and Geometric Computing, Technion
Matrix Class

- C++ Class for handling matrices and matrix operations

```cpp
class mat2 {
  vec2 _m[2];
public:
  mat2( const GLfloat d = GLfloat(1.0) ) {
    _m[0].x = d; _m[1].y = d;
  }
  mat2( const vec2& a, const vec2& b ) {
    _m[0] = a; _m[1] = b;
  }
  mat2( GLfloat m00, GLfloat m10, 
        GLfloat m01, GLfloat m11 ) {
    _m[0] = vec2( m00, m01 );
    _m[1] = vec2( m10, m11 );
  }
};
```
Matrix Class

- Operator overloading

```cpp
mat2 operator + ( const mat2& m ) const {
    return mat2( _m[0]+m[0], _m[1]+m[1] );
}
mat2 operator * ( const GLfloat s ) const {
    return mat2( s*_m[0], s*_m[1] ); }
mat2 operator * ( const mat2& m ) const {
    mat2  a( 0.0 );
    for ( int i = 0; i < 2; ++i )
        for ( int j = 0; j < 2; ++j )
            for ( int k = 0; k < 2; ++k )
                a[i][j] += _m[i][k] * m[k][j];
    return a;
}
```
Coordinate Systems

- A coordinate system in 3D is defined by 3 vectors $(v_1, v_2, v_3)$
- A vector is written $v = a_1 v_1 + a_2 v_2 + a_3 v_3$
- The list of scalars $(a_1, a_2, a_3)$ is the representation of $v$ with respect to the given basis

We can write in matrix form

$$v = \begin{pmatrix} v_x^1 & v_x^2 & v_x^3 \\ v_y^1 & v_y^2 & v_y^3 \\ v_z^1 & v_z^2 & v_z^3 \end{pmatrix} \begin{pmatrix} a_1 \\ a_2 \\ a_3 \end{pmatrix}$$

- Can coordinate systems be used to define position?
Right and Left Coordinate Systems

**Question:** What is a positive angle of rotation around an axis specified by a vector?

**Answer:** Align your thumb with the vector. Your other fingers indicate the direction of a positive rotation.

**Definition:** A three-dimensional left (right) coordinate system is such that the positive rotation direction around the Z axis using the left (right) thumb rotates X towards Y.

A three-dimensional *left* coordinate system.  
A three-dimensional *right* coordinate system.
Frames

- A coordinate system is insufficient to represent points.
- If we work in an affine space we can add a single point, the *origin*, to the basis vectors to form a frame.

Frame determined by \((P_0, v_1, v_2, v_3)\)
Coordinate Systems and Frames

- Model Frame
- World Frame
- View/Camera Frame
  - Projection Transform
- Screen Frame
- Device Frame
Model and World Frame

- The model coordinate system is where the object is defined
  - Customarily, objects are modeled around the origin
  - Why?
- The world coordinate system is where all the objects reside
  - To move objects from one coordinates system to the other we apply transformations on the object
  - Multiply the vertices of a polygonal object by a proper [4X4] matrix

Modeling Transform
The Abstract Model Class

- A base class for Models

```cpp
class Model {
protected:
    virtual ~Model() {} 
    void virtual draw()=0;
    Geometry T;
    mat4 mTransform;
};
```

- Geometry can be triangles, splines etc.
- `draw()` sends the renderer the geometry in world coordinates
- More about the renderer later
View/Camera Frame

- Frame that is centered on the camera
- Defined by the camera transform
- All objects in a scene are either in positive (in front of the camera) or negative (behind) Z
- Positions the viewing volume in the world
- Apply *inverse* of camera transform to objects in world frame to get them into view frame
  - Always maintain the inverse of the transformation
  - Remember: $(AB)^{-1} = B^{-1}A^{-1}$
The Model-View Matrix

- Define $M, C$ as the model and camera transformations in world frame
- The model-view matrix is $T = C^{-1}M$
  - Move from model frame to view frame
- There is a duality between $C$ and $M$
- We want to view an object from certain direction
  - We can move the camera,
  - Or we can move the object
The Model-View Matrix

\[ C = T_{1,-3}R_{45} \quad M = I \]
\[ T = (T_{1,-3}R_{45})^{-1}I \]
\[ T = R_{-45}T_{-1,3} \]

\[ C = I \quad M = R_{-45}T_{-1,3} \]
\[ T = IR_{-45}T_{-1,3} \]
\[ T = R_{-45}T_{-1,3} \]
The Camera Class

- A base class for cameras

```cpp
class Camera {
    // constructors
    mat4 cTransform;
    mat4 projection;

    public:
    void setTransformation(const mat4& T);
    void setProjection(const mat4& T);
    void LookAt(...);
    void Ortho(...);
    void Perspective(...);
    ...
};
```
The LookAt function

- LookAt is one way to set the camera transformation

\[
\text{mat4 LookAt(vec4& eye, vec4& at, vec4& up )}
\]

- eye – The position of the camera
- at – The position the camera looks at
- up – The upside (y) direction of the camera

- The three vectors used to define the camera frame
  - How?
The LookAt function

- LookAt is one way to set the camera transformation.

```cpp
mat4 LookAt(vec4& eye, vec4& at, vec4& up )
{
    vec4 n = normalize(eye - at);
    vec4 u = normalize(cross(up,n));
    vec4 v = normalize(cross(n,u));
    vec4 t = vec4(0.0, 0.0, 0.0, 1.0);
    mat4 c = mat4(u, v, n, t);
    return c * Translate( -eye );
}
```
The Scene Class

- A container for all objects, cameras, etc.

```cpp
class Scene {
    vector<Model*> models;
    vector<Camera*> cameras;
    renderer *m_renderer;

public:
    Scene(Renderer *renderer);
    void AddModel(Model* model);
    void AddCamera(Model* model);
    Model* GetModel(int model_id);
    ...
    void draw();
};
```
The Scene Class

- The Scene class contains all the information needed to describe the scene.

- When \texttt{draw()} is invoked, the scene tells all its objects to update the renderer:
  - Active camera sets the view frame and projection.
  - Models update their frame and submit the geometry.
  - And so on..
Screen and Device Frames

- Projecting the scene in camera frame gets us the screen frame.
- There are several options there as discussed in the lecture.
  - Orthographic, perspective, etc..
- It is preferred to consider normalized projected space where the projection transform maps to $[-1,1] \times [-1,1]$ square.

- To continue to device frame, coordinates must get integer values.
The Renderer Class

- Also called the device. Renders the scene to an output (the physical device)
  - Applies all transformation
  - Computes shading and effects
  - Creates a rasterization
class Renderer{
    float *m_outBuffer; // 3*width*height
    float *m_zbuffer; // width*height
    int m_width, m_height;
    void CreateBuffers(int width, int height);

public:
    //construction
    void Init();
    void DrawTriangles(vector<vec3>* vertices);
    void SetCameraTransform(mat4& cTransform);
    void SetProjection(mat4& projection);
    void SetObjectMatrix(mat4& oTransform,
                          mat3& nTransform);
    void SwapBuffers();
}
The Renderer Class

- void CreateBuffers(int width, int height);
  - Creates or resizes the frame buffers
- void DrawTriangles(vector<vec3>* vertices);
  - Draws all triangles to the color buffer taking every aspect of the scene into consideration
- void SetCameraTransform(mat4& cTransform);
  void SetProjection(mat4& projection);
  void SetObjectMatrix(mat4& oTransform, mat3& nTransform);
  - Set various matrices for the next drawing
- void SwapBuffers();
  - Loads the color buffer into the screen buffer
Basic Geometry
Line equations

What are the available forms

- Explicit form – \( f(x) = ax + b \)
- Implicit form – \( f(x, y) = ax + by + c = 0 \)
- Parametric form – \( f(t) = (1 - t)P_1 + tP_2 \)
Point-Line Distance

\[ P = (1 - t)P_1 + tP_2 \]

\[ \overrightarrow{PQ} = \frac{1}{2} \overrightarrow{QQ'} \]

\[ \| \overrightarrow{QQ'} \| \| \overrightarrow{P_1P_2} \| = \overrightarrow{QP_1} \times \overrightarrow{QP_2} \]

\[ \| \overrightarrow{QP} \| = \frac{1}{2} \| \overrightarrow{QQ'} \| = \frac{\overrightarrow{QP_1} \times \overrightarrow{QP_2}}{2 \| \overrightarrow{P_1P_2} \|} \]
Line-Line Distance

\[ f(t) = (1 - t)P_1 + tP_2 \]
\[ g(s) = (1 - s)Q_1 + tQ_2 \]

- The distance between \( f(t) \) and \( g(s) \) is
  \[ D(s, t) = \|f(t) - g(s)\| \]
- Minimize \( D^2(s, t) \) (Why and how?)

\[
\min D(s, t) = \frac{|(P_1 - Q_1) \cdot [(P_2 - P_1) \times (Q_2 - Q_1)]|}{|(P_2 - P_1) \times (Q_2 - Q_1)|}
\]

- What about segment-segment?
  - Need to check end points separately
Point-Plane Distance

\[ Ax + By + C + D = 0 \]
\[ w = (x_0 - x, y_0 - y, z_0 - z) \]
\[ n = (A, B, C) \]

- The distance is

\[ D = \frac{|w \cdot n|}{||n||} \]
\[ = \frac{|ax_0 + by_0 + cz_0 + d|}{\sqrt{a^2 + b^2 + c^2}} \]
Collision Detection
Collision Detection

- Check if and when moving object will collide
  - Physical simulations
  - Video games
  - Computational geometry

- Needs to be efficient and accurate
  - Real time
  - Not a stable problem

- Simpler problem - Collision of Static Primitives
  - Check if two primitives are intersecting
  - Much easier
    - Answer is only yes/no
    - Simple geometry
Example Line-Line Intersection

\[ L_1(t) = (1 - t)P_1 + tP_2, \quad L_2(r) = (1 - r)Q_1 + rQ_2 \]

Find \( r, t \) such that

\[ L_1(t) = (1 - t)P_1 + tP_2, = (1 - r)Q_1 + rQ_2 = L_2(r) \]

Two (why?) linear equations with two variables

**Question:** What is the meaning of \( r, t < 0 \) or \( r, t > 1 \)?
Example – Line-Sphere intersection

- The sphere equation is given by
  \[ D(P, P_0) = R \implies (x - x_0)^2 + (y - y_0)^2 + (z - z_0)^2 = R^2 \]

- Solve a quadratic system of equation
  - If solution exists, there is an intersection

- Is there a simpler way?
  - Find distance from line to center of sphere
  - If it is less than R, there is an intersection