GAME PROGRAMMING

Contents

MODULE 2

2D Graphics: Sprites, Tiled Images, and Backgrounds

Overview of 2D Graphics

- 2D graphics have been around since the very beginning of gaming.
- The way 2D images are generated and displayed has evolved significantly.
- Before the Nintendo 64 and PlayStation era, most console video games were created in 2D.
- Not all games that are simple have to be 2D, just like not all 2D games are simple.

Sprites

- 2D games use sprites for virtual objects and elements.
- A sprite is a 2D image drawn to the screen.
- In 3D games, sprites are typically used for particle effects.
- In 2D games, sprites are used for everything visible.

Types of 2D Sprites

- **Static Sprite**: A single sprite image that does not animate.
- **Dynamic Sprite**: A collection of images that form an animation, similar to a flip book.

Steps to Create and Use Sprites in Unity

Step 1: Creating Sprites

1. **Using an Image Editing Tool**: Adobe Photoshop, GIMP.
2. **Create the Sprite**:
    - Open the image editing tool.
    - Create a new image with a transparent background.
    - Draw or import the 2D graphic.
    - Save the image as a PNG file to preserve transparency.

Step 2: Importing Sprites into Unity

1. **Open Unity**:
    - Start Unity and create a new project or open an existing one.

2. **Import the Sprite**:
    - Drag and drop your PNG file into the Assets folder in the Unity Editor.
3. **Configure the Sprite**:
    - Select the imported image in the Assets folder.
    - In the Inspector window, set the Texture Type to Sprite (2D and UI).
    - Select Compression and click Apply.

Step 3: Using Sprites in Unity

1. **Create a New Scene**:
    - Open or create a new scene in Unity.
2. **Add the Sprite to the Scene**:
    - Drag the sprite from the Assets folder into the Hierarchy window or directly into the Scene view.
3. **Adjust Position and Scale**:
    - Use the Transform component in the Inspector window to position and scale the sprite as needed.

Step 4: Animating Sprites

1. **Creating a Sprite Sheet**:
    - Combine all frames of animation into a single image file laid out in a grid.
    - Import the sprite sheet PNG file into Unity.
2. **Slice the Sprite Sheet**:
    - Select the sprite sheet in the Assets folder.
    - Change Sprite Mode to Multiple, Pixels per unit to 48, and Filter mode to Point.
    - Click the Sprite Editor button, choose Slice, set parameters, and click Slice and Apply.

Step 5: Creating an Animation

1. **Select the Sliced Sprites**:
    - In the Assets folder, expand the sprite sheet to reveal the individual frames.
    - Select all frames for the animation.
2. **Create an Animation Clip**:
    - Drag the selected frames into the Scene or Hierarchy window.
    - Name and save the animation clip.
3. **Animator Controller**:
    - Unity will create an Animator Controller automatically.
    - Ensure the sprite has an Animator component with the created Animator Controller assigned.

Step 6: Scripting for Interaction

1. **Create a Script**:
    - In the Assets folder, right-click and select Create > C# Script.
    - Name it SpriteController.

```
using UnityEngine;

public class SpriteController : MonoBehaviour
{
    public float moveSpeed = 5f;
    private Animator animator;

    void Start()
    {
        animator = GetComponent<Animator>();
    }

    void Update()
    {
        float moveX = Input.GetAxis("Horizontal");
        float moveY = Input.GetAxis("Vertical");
        Vector2 move = new Vector2(moveX, moveY);

        if (move != Vector2.zero)
        {
            animator.SetBool("isMoving", true);
            transform.Translate(move * moveSpeed * Time.deltaTime);
        }
        else
        {
            animator.SetBool("isMoving", false);
        }
    }
}
```

2. **Attach the Script to the Sprite**:
    - Select the sprite in the Hierarchy window.
    - In the Inspector window, click Add Component and attach the SpriteController script.
3. **Configure the Animator**:
    - Open the Animator window (Window > Animation > Animator).
    - Create a parameter named isMoving (type: Bool).
    - Set up transitions between idle and moving animations based on the isMoving parameter.

Step 7: Testing

1. **Play the Scene**:
    - Click the Play button to run the scene.

- Use the arrow keys or WASD keys to move the sprite and see the animation.

Tiled Images and Backgrounds

Overview

- In 2D game graphics, sprites are used together to create environments and backgrounds.
- One or more sprite images often act as the environment's background (e.g., sky, clouds, underwater, etc.).
- The background may scroll with the player to give the effect of moving through the environment.

Foregrounds Using Tiled Sprites

- In addition to the background, there is often a foreground environment.
- This foreground environment is typically made up of tiles.
- A tile is an image (sprite) composed of a repeatable pattern.
- By placing tile-able images next to each other, artists can create complex environments from a small image set.

Tiled Images and Backgrounds

- **Tiled Images**:
  - Ideal for 2D games with large, scrolling environments.
  - Suitable for procedurally generated levels or maps.
  - Important for memory efficiency and scalability.
  - Provides flexibility in designing and modifying environments.
- **Background**:
  - Best for scenes requiring detailed, custom artwork.
  - Suitable for static scenes or smaller, fixed environments.
  - Prioritizes artistic detail and uniqueness over scalability.
  - Simpler implementation without managing multiple tiles.

Steps to Create and Use Tiled Images in Unity

Step 1: Creating Tiled Images

- **Create a Seamless Texture**:
  - Use an image editing tool like Adobe Photoshop, GIMP, or Aseprite.
  - Design a texture that tiles seamlessly (edges match perfectly when repeated).
  - Save the texture as a PNG file.

Step 2: Importing the Texture into Unity

1. **Open Unity**:
   - Start Unity and create a new project or open an existing one.

2. **Import the Texture**:
    - Drag and drop the PNG texture file into the Assets folder in the Unity Editor.
3. **Configure the Texture**:
    - Select the imported texture in the Assets folder.
    - In the Inspector window:
        - Set the **Texture Type** to **Sprite (2D and UI)** for 2D games.
        - Ensure **Wrap Mode is set to Repeat** to enable tiling.
        - Click Apply.

Step 3: Using Tiled Textures in Unity

1. **Create a Quad or Plane**:
    - In the Hierarchy window, right-click and select 3D Object > Quad or Plane.
    - This object will serve as the base for the tiled background.
2. **Create a Material**:
    - In the Assets folder, right-click and select Create > Material.
    - Name the material (e.g., TiledBackgroundMaterial).
3. **Apply the Texture to the Material**:
    - Select the created material.
    - In the Inspector window, ***assign the texture*** to the ***Albedo*** property.
4. **Configure Tiling**:
    - In the Inspector window of the material, ***adjust the Tiling settings*** under the ***Main Maps section*** to control how the texture repeats.
    - Set ***Tiling X*** and ***Tiling Y*** to control the number of times the texture repeats horizontally and vertically.
5. **Apply the Material to the Quad/Plane**:
    - Drag the material onto the Quad or Plane in the Scene or Hierarchy window.

Step 4: Using Tiled Textures for 2D Backgrounds

1. **Create a Sprite Renderer**:
    - In the Hierarchy window, right-click and select 2D Object > Sprite.
    - This object will serve as the base for the tiled background.
2. **Create a Material**:
    - In the Assets folder, right-click and select Create > Material.
    - Name the material (e.g., TiledBackgroundMaterial).
3. **Apply the Texture to the Material**:
    - Select the created material.
    - In the Inspector window, assign the texture to the ***Main Texture*** property.

4. **Configure Tiling**:
    - In the Inspector window of the material, adjust the Tiling settings under the ***Main Maps section*** to control how the texture repeats.
    - Set ***Tiling X*** and ***Tiling Y*** to control the number of times the texture repeats horizontally and vertically.
5. **Apply the Material to the Sprite**:
    - Select the Sprite object in the Hierarchy window.
    - In the Inspector window, change the Sprite Renderer component's ***Material property*** to the created material.

Step 5: Scripting for Dynamic Tiling (Optional)

1. **Create a Script**:
    - In the Assets folder, right-click and select Create > C# Script.
    - Name it DynamicTiling.
2. **Attach the Script to the Object**:
    - Select the Quad/Plane or Sprite in the Hierarchy window.
    - In the Inspector window, click Add Component and attach the DynamicTiling script.

Step 6: Testing

1. **Play the Scene**:
    - Click the Play button to run the scene.
    - Observe the tiled texture repeating on the Quad/Plane or Sprite as expected.

3D GRAPHICS

- One of the ***main focuses of 3D game development*** is computer graphics.
- Often, the ***goal*** is to achieve ***realistic real-time graphics*** in games, as seen in *Alan Wake*.

The Z Buffer

- In modern 3D games, one of the important pieces of information used during the ***rendering*** process is ***depth information***.
- Depth information, ***stored in the Z buffer***, informs the ***rendering API*** about the calculated ***depth of each pixel*** on the ***screen***.
- This information is mainly used to ***determine what objects are in front*** of others.
- This process is commonly known as ***depth testing***.
- In ***2D games***, ***sprites*** are simple enough to be given a ***draw order***, and objects are drawn in the specified order.
- In ***3D games***, this is not as simple, as surfaces are not 2D and can have ***varying positions*** and ***orientations***.

- ***Neighboring pixels*** can have ***different depths*** across even a ***single*** surface.
- When it comes to ***lighting*** and other common rendering techniques in 3D, it is not possible to use one value for an entire surface as you can with 2D, assuming the surface isn't perfectly flat and facing the camera.

Shading Surfaces

- One of the ***main advantages 3D games*** have over 2D games is the ability to ***dynamically calculate surface information*** in the scene.
- By shading the surfaces that compose the scene, ***developers can perform many calculations*** that ***affect the final result*** dynamically.
- One of the best examples of this is real-time lighting. By using the information in the scene, we can shade each point/fragment that makes up the geometry, giving it a ***realistic appearance*** and ***interactions***.
- The shading of surfaces is the ***entire point*** of 3D computer graphics.

GEOMETRY AND PRIMITIVES

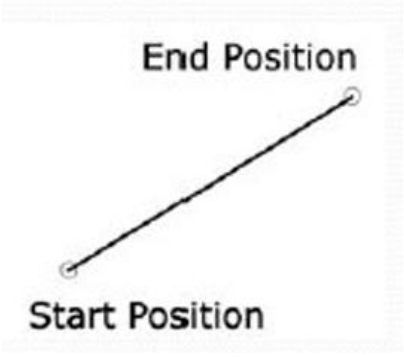
- The ***heart of 3D*** lies in using various geometric entities to represent the ***objects and environments*** of a virtual scene.
- In 3D video games, this information is usually processed by a dedicated piece of hardware: the ***graphics card***.
- The ***rendering pipeline*** can be thought of as a ***series of algorithmic steps*** that operate on the ***data*** that make up the ***scene to be rendered***.

Lines

- A line is a ***simple primitive*** that has a ***starting location*** and an ***ending location***. With two points, a line segment can be represented in 2D or 3D space.

```
struct Point {
    int x;
    int y;
};

struct Line {
    Point start;
    Point end;
};
```



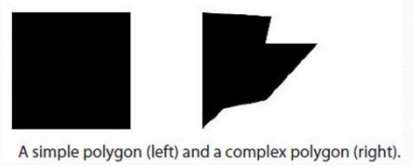
- Lines can be used to form ***mathematical rays*** where the ***starting*** point of a line is the ***ray's origin*** and the ***ending*** point determines the ray's ***direction***.

Polygons

- Polygons form many of the ***geometric shapes*** seen in today's 3D games.
- A polygon is a geometric shape made up of ***three or more points***, where the lines connecting the outer area of the shape are called ***edges***.

- The ***area*** within this shape is often ***filled*** using various ***algorithms*** (e.g., ***lighting, texturing mapping***) by the hardware assigned the rendering task.

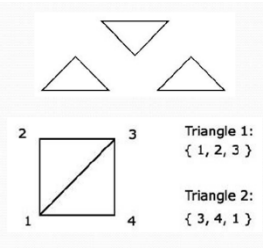
```
struct Polygon {
    int total_points;
    array<Point> points;
};
```


A simple polygon (left) and a complex polygon (right).

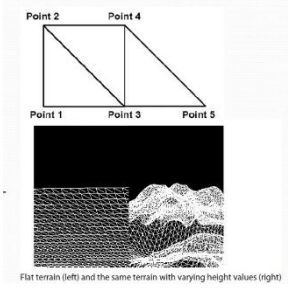
- The ***more points*** that are used, the ***more complex*** a shape can look.

Triangles

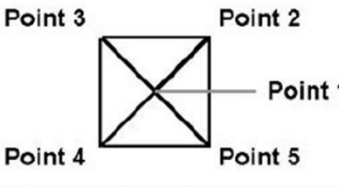
- Triangles are the ***most common type of primitive*** used in 3D video games.
- Triangles are ***three-point polygons*** whose three edges connect each of the points that make up the shape of the primitive.
- ***Graphics hardware*** is ***very efficient*** at ***processing*** triangle primitives.
- Games often use three types of triangles.
- **Triangle Lists**: ***Individual triangles*** specified in an ***array***, sometimes referred to as a ***buffer***. Each triangle is individually specified, and ***points can be shared by using indices*** to reduce the amount of data passed to the graphics hardware.



- An example of using indices is shown, where four points are used to specify two distinct triangles.
- **Triangle Strips**: Defined by specifying the fi***rst three points of the first triangle*** and then ***one point*** for each ***additional*** triangle that branches off the first. This reduces the amount of data passed to the graphics hardware.



- By using six points, we can form four separate triangles. Triangle strips are commonly used for ***terrain geometry***, where a terrain is normally a ***rectangular grid with varying Y-axis (height) values for each polygon***.
- The last type of triangle is **Triangle Fans**, which are triangles that all ***connect to the same common point on the mesh***. By specifying a common point that all triangles connect to, you can create various shapes that would require more information to create with triangle lists.



Convex and Concave Polygons

- Polygons and shapes in computer graphics can be either convex or concave.
- A ***convex polygon***, or shape, has a convex set for its surface, meaning that if a ***line segment*** is created between any two points that make up the object, it ***cannot penetrate*** any edge of the ***object***.
- If a line segment does penetrate one of the object's edges, it is considered ***concave***.

- Using ***convex geometry is a better*** choice for things such as ***collision detection*** because convex shapes and polygons are more ***efficient to calculate*** and work with ***mathematically*** on a processor.
- A ***strict convex polygon*** or shape is one that has ***at most a 180-degree*** angle ***between each edge*** that makes up the object.

Spheres and Boxes

- Spheres and boxes are commonly used in video games, not just for ***rendering*** but also for ***physics and collision calculations***.
- A ***sphere*** is a mathematical object with a ***position*** and a ***radius***, which specifies the ***circular region that surrounds the position***.

```
struct Sphere {
    int radius;
    Point position;
};
```

- Spheres and boxes are used to ***surround complex objects*** in virtual scenes.
- If a complex object, such as a character model, is surrounded with a sphere or box primitive, they can be used in ***collision and physics tests*** as a fast way to determine what action needs to be taken.
- For example, the ***test between two spheres colliding*** in a virtual scene is much ***faster than testing two triangles***. By testing two simple spheres, you can quickly ***determine if a collision is even possible*** between two complex character models.
- When it comes to collision detection against spheres, ***spheres are much faster*** than other basic primitives such as triangles, boxes, and so forth. This speed comes at a ***cost of accuracy*** because the region that surrounds most objects tends to have a lot of ***wasted space***.
- Using models with hundreds or thousands of polygons, the CPU power required to calculate the average game physics using triangle-to-triangle tests would be so vast that games like *Halo 3*, *Assassin's Creed*, *Metal Gear Solid 4*, and even *Super Mario 64* (during its time) would be impossible.
- The ***triangle-to-triangle collision*** test is so ***expensive*** that game developers avoid them altogether and instead perform collisions on groups of very simple shapes.

Additional Geometric Objects

- Cones
- Pyramids
- Cylinders
- Toruses
- Torus knots
- Disks

- Ellipsoids
- Bezier curves
- Bezier patches
- NURBS

Objects found in video games are often ***created using triangles*** rather than some mathematical equation.

Loading Geometry

In video games, geometry is usually loaded into an application during runtime from an ***external file***. Games use many types of files like:

- Static models
- Animation files
- Terrains
- Environment files
- Collision meshes

MATHEMATICS USED IN COMPUTER GRAPHICS

- ***Games graphics*** are made up of a ***lot of mathematics***.
- The mathematics used in game graphics, physics, and so forth can become quite ***complex and advanced***.
- Having a firm understanding of the different types of mathematics allows you to have an easier time understanding and implementing the information.

Vectors, Vertices, and Points

- Vectors are the fundamental mathematical objects used in every 3D game and game engine.
- Vectors define a direction in virtual space, but they can also be used to define points called vertices. For example, a triangle is made up of three of these points.
- Technically, vectors and vertices are different, but they are used the same way most of the time in games. Vectors are spatial directions, and vertices are points of a primitive.

Vectors, Vertices, and Points (contd)

- Vectors come in many types, with the most common ones being 2D, 3D, and 4D.
- A vector is made up of n number of dimensions that describe the total number of axes it uses.
    - For example, a 2D vector only has an X and Y axis.
    - A 3D vector has an X, Y, and Z axis.

    - A 4D vector has the same axes as a 3D vector in addition to a W axis.
- A vector can generally be written as V = (V1, V2, …, Vn).

Vectors, Vertices, and Points (contd)

In a language such as C or C++, a 3D vector can have the following structure:

```
struct Vector3D {
    float x, y, z;
};
```

- Vectors can be operated on by scalars, which are floating-point values.

```
V.x = V1.x + V2.x;
V.y = V1.y + V2.y;
V.z = V1.z + V2.z;

V.x = V1.x + A;
V.y = V1.y + A;
V.z = V1.z + A;
```

- For example, you can add, subtract, multiply, and divide a vector with another vector or a scalar.

Additional Vector Operations

- **Normalization**: A unit vector from a non-unit vector.

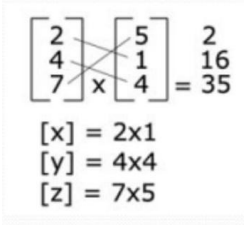
$$\text{Magnitude} = \text{Square\_Root}(V.x^2 + V.y^2 + V.z^2)$$

$$V = V \,/\, \text{Square\_Root}(V.x^2 + V.y^2 + V.z^2)$$

- **Dot Product**: The dot product of two vectors, also known as the scalar product, calculates the difference between the directions the two vectors are pointing.
    - The dot product is used to calculate the cosine angle between vectors without using the cosine mathematical formula, which can be more CPU expensive.
    - The result is a scalar value.

$$d = (V1.x * V2.x + V1.y * V2.y + V1.z * V2.z)$$

    - If the dot product between two vectors is 0, the two vectors are orthogonal (perpendicular to one another).
    - The sign of the dot product tells us:
        - If the sign is negative, the second vector is behind the first.
        - If it is positive, then it is in front.
        - If it is 0, they are perpendicular.
- **Cross Product**: The cross product of two vectors, also known as the vector product, is used to find a new vector that is perpendicular to two tangent vectors.



Transformations

- 3D computer graphics incorporate the idea of many different coordinate spaces.
- A coordinate space represents an object's relationship to the rest of the scene. For example, the vertices of a 3D model are often stored in object-space, which is a space local to the model itself.
- To render out an object that is in object-space, it must be transformed to the world-space position at which you want to render it.
- When rendering objects in a 3D scene, we use positions and orientations to represent how an object is located.
- This information is used to create a mathematical matrix that can be used to transform the vertex data of rendered geometry from one space to another.
- The positions and orientations are specified in world-space, which is also known as model-space.
- Once an object in object-space is transformed to world-space, it is transformed to screen-space, which corresponds to the X and Y axes that are aligned to the screen.

Projections

- Since the 3D information has depth and distance with objects that are farther from the camera, a projection is applied to the geometry to add perspective to the data.
- The projection matrices that are used on geometry are called homogeneous clip space matrices, which clip the geometry to the boundaries of the screen to which they are being rendered.
- Two types of projection are generally used in video games: orthogonal projection and perspective projection.

Projection Overview

- When you render a 3-dimensional computer graphics scene, you create a 2-dimensional picture of the 3D scene. The picture is a projection of the models in the scene onto a 2-dimensional "screen".
- Therefore, it is logical to call this operation in the graphics pipeline a projection.
- There are two standard projections used in computer graphics:
    - **Orthogonal projection**: Maintains parallel lines but provides no sense of depth.

    - **Perspective projection**: Provides for a sense of depth, but parallel lines are skewed toward vanishing points.
- Orthographic projections are used in the engineering fields when an accurate representation of a model is desired.
- Perspective projections are used when a “real life” view of a scene is desired, simulating how the human eye sees the real world.

Orthogonal Projection

- Orthogonal projection maps a 3D object to a 2D view, but objects remain the same size regardless of their distance from the camera.
- An example of an orthographic projection:
    - Parallel lines stay parallel.
    - There is no perception of depth.

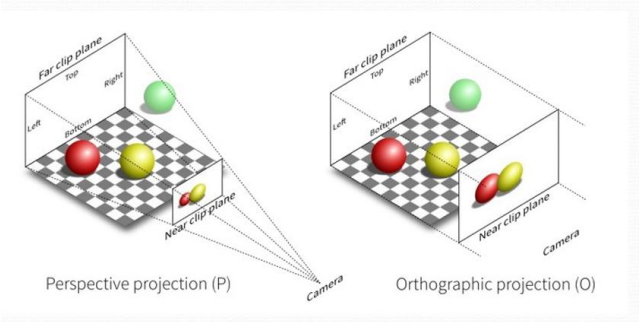
[Orthographic Projection Example]

Perspective Projection

- In perspective projection, perspective is added to the rendered scene, which makes objects smaller as they move farther from the camera.
- An example of a perspective projection:
    - Parallel lines of the model are not parallel in the rendering.
    - You can perceive depth (the off-center object changes size as its distance from the camera changes).

[Perspective Projection Example]

Projections in a Frustum



Matrices

- A matrix is a mathematical structure used in computer graphics to store information about a space.
- In computer graphics, matrices are often used for storing orientations, translations, scaling, coordinate spaces, and more.

```
struct Matrix
{
    Vector3D col1;
    Vector3D col2;
    Vector3D col3;
}

struct Matrix
{
    Vector3D mat[3];
}
```

- A matrix is essentially a table, for example:

```
float matrix3x3[3][3];
matrix3x3[0] = 1; matrix3x3[1] = 0; matrix3x3[2] = 0;
matrix3x3[3] = 0; matrix3x3[4] = 1; matrix3x3[5] = 0;
matrix3x3[6] = 0; matrix3x3[7] = 0; matrix3x3[8] = 1;
```

- A matrix can be considered an array of vectors that together represent a space.
- When it comes to orientations in video games, a matrix is used to store rotational and positional information along with scaling.

Transformations in 3D

- Transformations in 3D graphics involve changing the position, orientation, and scale of objects. These transformations are achieved using matrices and are applied in a specific order to achieve the desired effect.

Types of Transformations

1. **Translation**:
    - **Purpose**: Moves an object from one location to another in 3D space.
    - **Matrix Representation**: To move a point P with coordinates (x0, y0, z0) to a new point P’ in 3D space, multiply the translation matrix T by the point's homogeneous coordinate vector.

$$T(x,y,z) = \begin{bmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$P' = T \cdot P$$

$$P' = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x + t_x \\ y + t_y \\ z + t_z \\ 1 \end{bmatrix}$$

$$P = \begin{bmatrix} x_0 \\ y_0 \\ z_0 \\ 1 \end{bmatrix}$$

    - The new point P’ is obtained by:

$$P' = T(x,y,z) \cdot P$$

$$P' = \begin{bmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x_0 \\ y_0 \\ z_0 \\ 1 \end{bmatrix}$$

$$P' = \begin{bmatrix} 1 \cdot x_0 + 0 \cdot y_0 + 0 \cdot z_0 + x \cdot 1 \\ 0 \cdot x_0 + 1 \cdot y_0 + 0 \cdot z_0 + y \cdot 1 \\ 0 \cdot x_0 + 0 \cdot y_0 + 1 \cdot z_0 + z \cdot 1 \\ 0 \cdot x_0 + 0 \cdot y_0 + 0 \cdot z_0 + 1 \cdot 1 \end{bmatrix}$$

    - Thus, the new coordinates (x’, y’, z’) are:

$$x' = x_0 + x$$
$$y' = y_0 + y$$
$$z' = z_0 + z$$

$$P' = \begin{bmatrix} x_0 + x \\ y_0 + y \\ z_0 + z \\ 1 \end{bmatrix}$$

- The point (x0, y0, z0) is moved by x units along the x-axis, y units along the y-axis, and z units along the z-axis.

Types of Transformations

1. **Rotation**: Rotates an object around an axis in 3D space.
    - **Matrix Representation**:
        - **Around the X-axis**:

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

        - **Around the Y-axis**:

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

        - **Around the Z-axis**:

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

Applying a Rotation Matrix

- To rotate a point P with coordinates (x0, y0, z0) around an axis, multiply the corresponding rotation matrix by the point's homogeneous coordinate vector.

$$P = \begin{bmatrix} x_0 \\ y_0 \\ z_0 \\ 1 \end{bmatrix}$$

- **Example**: Rotating around the X-axis by an angle θ.

$$P' = R_x(\theta) \cdot P$$

$$P' = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta & 0 \\ 0 & \sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x_0 \\ y_0 \\ z_0 \\ 1 \end{bmatrix}$$

$$P' = \begin{bmatrix} 1 \cdot x_0 + 0 \cdot y_0 + 0 \cdot z_0 + 0 \cdot 1 \\ 0 \cdot x_0 + \cos\theta \cdot y_0 + -\sin\theta \cdot z_0 + 0 \cdot 1 \\ 0 \cdot x_0 + \sin\theta \cdot y_0 + \cos\theta \cdot z_0 + 0 \cdot 1 \\ 0 \cdot x_0 + 0 \cdot y_0 + 0 \cdot z_0 + 1 \cdot 1 \end{bmatrix}$$

$$P' = \begin{bmatrix} x_0 \\ y_0 \cos\theta - z_0 \sin\theta \\ y_0 \sin\theta + z_0 \cos\theta \\ 1 \end{bmatrix}$$

Types of Transformations (contd)

2. **Scaling**: Changes the size of an object.
    - **Matrix Representation**:

$$S(x,y,z) = \begin{bmatrix} x & 0 & 0 & 0 \\ 0 & y & 0 & 0 \\ 0 & 0 & z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Occlusions

- An **occluder** is an object that blocks the sight of another object or objects.
- When performing **occlusion culling**, the idea is to determine which objects in the visible view frustum are blocked by other objects.
- By performing this test quickly on key objects, we can avoid rendering objects that we cannot see.
- By enclosing the view volume, we can determine if an object is visible or not. Objects that are not visible are not sent down the rendering pipeline, whereas those that are visible are drawn.
- This can be a huge optimization if we are able to use a frustum to cull a large number of polygons from the rendering process. If enough polygons are quickly culled out, the performance benefits can become apparent early on in the development of a game environment.
- This technique is commonly called **frustum culling** and is a very effective method used to speed up the rendering of a complex scene.

Unity Code Example

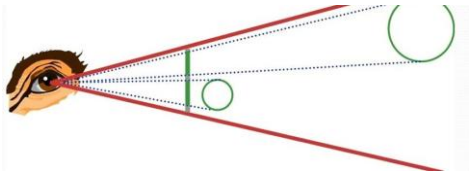
```
using UnityEngine;

public class TransformExample : MonoBehaviour
{
    void Start()
    {
        transform.position = new Vector3(1, 2, 3); // Initial position
        transform.position += new Vector3(3, 4, 5); // Translate by (3, 4, 5)
        transform.Rotate(new Vector3(0, 0, 90)); // Rotate 90 degrees around Z-axis
        transform.localScale = new Vector3(2, 2, 2); // Scale by (2, 2, 2)
    }
}
```

Transformations Overview

- **Modeling Transforms**: Size, place, scale, and rotate objects parts of the model with respect to each other (object coordinates, world coordinates).
- **Viewing Transform**: Rotate & translate the world to lie directly in front of the camera. Typically, place the camera at the origin looking down the Z-axis (world coordinates, view coordinates).

- **Projection Transform**: Apply perspective foreshortening. ***Distant*** objects appear ***smaller*** (the ***pinhole camera model***) (view coordinates, screen coordinates).



Transformations (contd)

- All these transformations involve shifting coordinate systems (i.e., basis sets).
- Represent coordinates as vectors, transforms as matrices.

$$\begin{bmatrix} X' \\ Y' \end{bmatrix} \begin{bmatrix} \cos\theta & -\sin\theta & N \\ \sin\theta & \cos\theta & Y \end{bmatrix}$$

- Multiply matrices = concatenate transforms!

Homogeneous Coordinates

- Homogeneous coordinates represent coordinates in 3 dimensions with a 4-vector denoted as **[x, y, z, w]**^{T}.
- ***w = 1 in model coordinates***.
- To get 3-D coordinates, divide by w:

$$[x', y', z']^T = [x/w, y/w, z/w]^T$$

- Transformations are represented as 4x4 matrices.

Quaternion

- Quaternions are a ***mathematical representation of rotations*** in ***3D space*** that are used in computer graphics for a variety of applications.
- Quaternions ***extend complex numbers to higher dimensions***, particularly useful in 3D mathematics for representing ***rotations and orientations***.
- A quaternion is a ***four-dimensional vector*** and can be written in the form:

q = w + xi + yj + zk

where w, x, y, and z are real numbers, and ***i, j, and k*** are the fundamental ***quaternion units***.

- $i^2 = j^2 = k^2 = ijk = -1$
- $ij = k$
- $ji = -k$
- $jk = i$
- $kj = -i$
- $ki = j$
- $ik = -j$

Quaternion Operations

1. **Addition:**

$$q_1 + q_2 = (w_1 + w_2) + (x_1 + x_2)i + (y_1 + y_2)j + (z_1 + z_2)k$$

2. **Multiplication:**

$$q_1 \cdot q_2 = (w_1w_2 - x_1x_2 - y_1y_2 - z_1z_2) + (w_1x_2 + x_1w_2 + y_1z_2 - z_1y_2)i + (w_1y_2 - x_1z_2 + y_1w_2 + z_1x_2)j + (w_1z_2 + x_1y_2 - y_1x_2 + z_1w_2)k$$

3. **Conjugate:**

$$q^* = w - xi - yj - zk$$

4. **Norm :**

$$\|q\| = \sqrt{w^2 + x^2 + y^2 + z^2}$$

Rays

- Rays are simple to represent in code. A ray is made up of a point of origin (position) and a direction.

```
struct Ray
{
   Vector3D origin;
   Vector3D direction;
};
```

- A ray can be thought of as an infinite line.
- Scalar values can be used to limit the distance of a ray.
- **Ray casting**, which is a technique used to test objects against a ray, can be used for:
    - Testing if one object can see another (line-of-sight)
    - Collision detection
    - Rendering
    - User interactions

Planes

- A plane is an infinitely flat surface that expands indefinitely across two axes.
- Planes can be thought of as flat walls that have a height and width.

$$Ax + By + Cz + D = 0$$

```
struct Plane
{
    Vector3D point;
    Vector3D direction;
    float d;
};
```

- Planes have many uses in game development, such as collision detection and culling for objects that are not in the camera's sight.
- The point specifies some point that lies on the infinite plane.
- The direction specifies the direction the plane is facing.

Quaternion

- Quaternions are useful for representing rotations in 3D space. A unit quaternion can represent a rotation.
- A rotation quaternion can be expressed as:

q = cos(θ/2) + sin(θ/2) * (x*i + y*j + z*k)

where **θ** is the ***rotation angle*** and (***x, y, z***) is the ***unit vector*** representing the axis of rotation.

To Rotate a Vector

- To rotate a vector v using a quaternion q, follow these steps:

1. Convert $\mathbf{v}$ to a pure quaternion $v_q = 0 + v_x i + v_y j + v_z k$.
2. Compute the rotated quaternion using:
   $v_q' = q \cdot v_q \cdot q^{-1}$
3. Extract the vector part of v_q' to get the rotated vector.

Rotate a vector v=(1,0,0) by 90 degrees around the z-axis

1. Create the Rotation Quaternion:

- Rotation angle θ=π/2
- Axis of rotation: (0,0,1)

The rotation quaternion is:

q=cos((π/2)/2)+sin((π/2)/2)(0i+0j+1k)

q=cos(π/4)+sin(π/4)(0i+0j+1k)

q= $\sqrt{2}/2$ + $\sqrt{2}/2$ k

- Convert the Vector to a Quaternion:

  vq=0+1i+0j+0k

- Calculate the Conjugate of $q*$=$\sqrt{2}/2$ + $\sqrt{2}/2$ k
- Apply the Rotation:

$v_q' = q \cdot v_q \cdot q^*$

Perform the quaternion multiplication to get:

$v_q' = \left(\frac{\sqrt{2}}{2} + \frac{\sqrt{2}}{2}k\right) \cdot (1i) \cdot \left(\frac{\sqrt{2}}{2} - \frac{\sqrt{2}}{2}k\right)$

Simplifying this, you get:

$v_q' = 0 + 0i + (-1)j + 0k$

Which is:

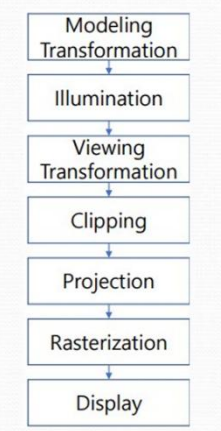
$v' = (0, -1, 0)$

Rendering Pipeline

- **Graphics (Rendering) Pipeline**:
  - **Objective**: Draw virtual objects (3D) on your screen (2D).
  - Commonly used for real-time applications.
  - Consists of multiple transformations.

Traditional Rendering Pipeline

- **Input**: Geometric model (e.g., primitives).
- **Output**: Colors (e.g., 24-bit RGB value at each pixel).



Modeling Transformation

- 3D models are defined in object space.
- Usual modeling tools: 3DS MAX and Maya (commercial), Blender (free to use).
- Where do we get 3D models? (e.g., pbrt.org and many websites for research, TurboSquid for commercial use).
- 3D models are defined in object spaces.
- We usually want to render a scene that contains multiple objects, needing to arrange all 3D models in a unique space (world space).
- In the world space: All 3D models, light sources, and the camera.
- Why do we need to use modeling transformation?

Illumination

- Illuminate 3D objects according to lighting and reflectance.
- Generally define materials of each object when designing models.

Lighting

- Illuminating a scene: Coloring pixels according to some approximation of lighting.
  - **Global Illumination**: Solves for lighting of the whole scene at once.
  - **Local Illumination**: Local approximation, typically lighting each polygon separately.
- Interactive graphics (e.g., hardware) does only local illumination at runtime.

Viewing Transformation

- Transform all points from world space to eye space.

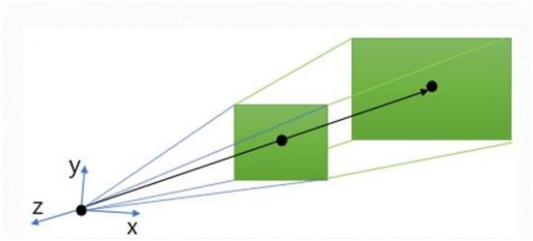
- Camera position transforms into the origin.

The Viewing Pipeline

- **Window**: Area selected in world-coordinate for display is called the window. It defines what is to be viewed.
- **Viewport**: Area on a display device in which the window image is displayed (mapped) is called the viewport. It defines where to display.
- In many cases, the window and viewport are rectangles, although other shapes may be used.
- Finding device coordinates of the viewport from world coordinates of the window is called viewing transformation.
- Sometimes considered as window-to-viewport transformation, but generally involves more steps.

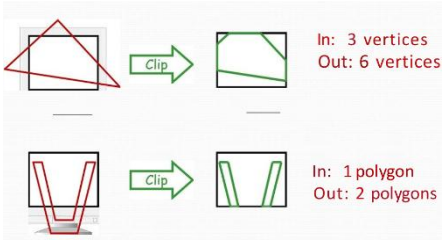
Clipping and Projection

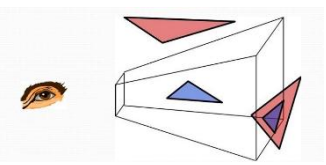
- A volume, viewing frustum, is specified from the camera.
- Map the frustum to the unit cube.
- Clip objects against the volume (remove non-visible geometry from your eye).
- Project objects into the 2D plane.
- Transform from eye space to normalized device coordinates.



Clipping

- Clipping a 3-D primitive returns its intersect ion with the view frustum:
  - **In**: 3 vertices
  - **Out**: 6 vertices
  - **In**: 1 polygon
  - **Out**: 2 polygons



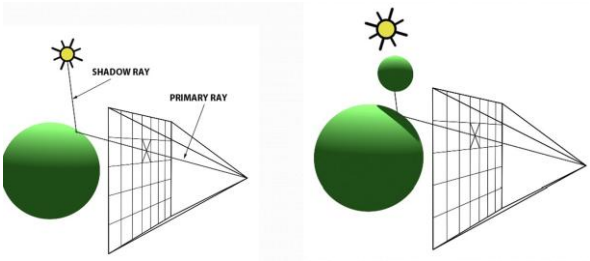


Rasterization and Display

- Transform normalized device coordinates to screen space.
- Rasterize the objects to fill color values at pixels.
- Most components in the graphics pipeline are transformations.

Rays (contd)

- Rays can also be used for rendering a scene image.
- In **ray tracing**, the idea is to create and cast multiple rays from all light sources into a scene going in different directions.
    - If a ray hits an object in the scene and if that object is in the camera's view, then that point of intersection is recorded.
    - Launches a primary ray into the scene by drawing a line from the eye through the pixel's center.
    - Ascertains if it intersects with any scene objects.
    - Selects the intersection nearest to the eye for further processing.
    - A secondary ray, known as a shadow ray, is then projected from this nearest intersection point towards the light source.
    - If it intersects another object en route, it signifies the casting of a shadow on the initial point.



Rays (final note)

- In ray tracing, you only record the information of the closest intersection.

The Scene Graph

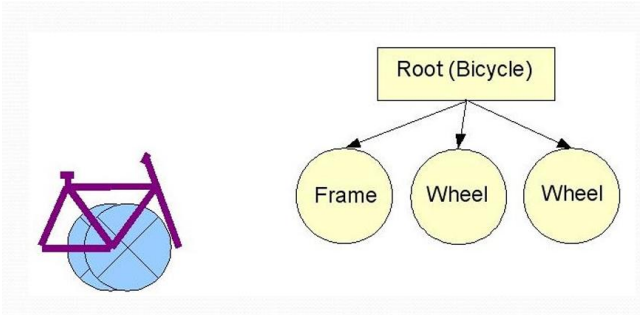
- A scene graph is a data structure commonly used by 3D rendering applications.
- The scene graph is the core data structure around which a graphics engine is built.
- The scene graph is a structure that arranges the logical and spatial representation of a graphical scene.
- Scene graphs are a collection of nodes in a graph or tree structure, normally a directed acyclic graph (DAG).
- The effect of a parent is apparent to all its child nodes; an operation applied to a group automatically propagates its effect to all of its members.

Modeling: Scene Graph

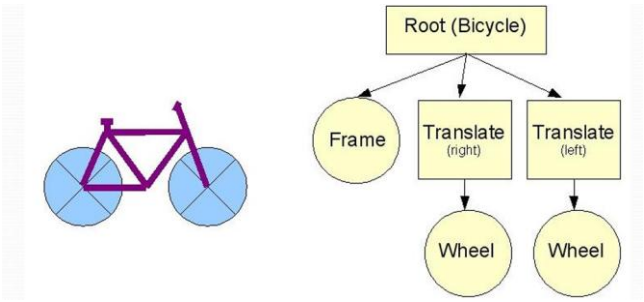
- Scene graphs normally employ two types of nodes:
    - **Leaf Node**: (no children) are normally actual renderable objects, elements of geometry (e.g., spheres, cubes, tori, and more complex models imported from 3D model design applications).
    - **Group Node**: (may have one or more children) are normally used to control state changes, color, transformations, materials, and animations.

Example for Scene Graph

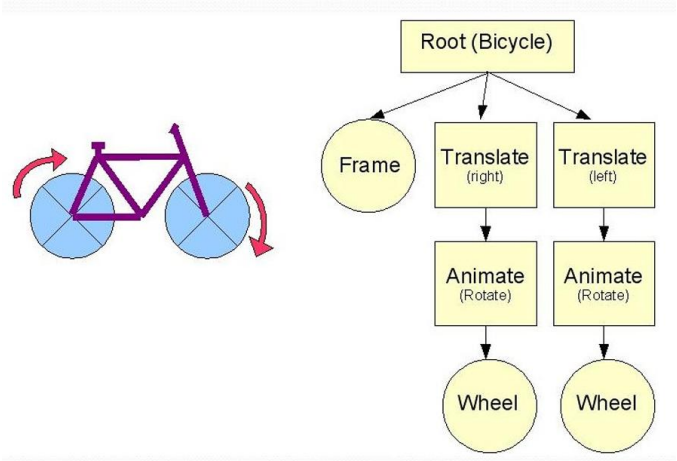
- Consider a bicycle made up of a frame and two wheels. This situation could be represented as follows (assuming we have some rendering code for the frame and wheels).



- This image isn't quite right. Most objects and elements of models are drawn at the origin. To move the wheels to their correct position, we need to add transformation nodes to the scene.

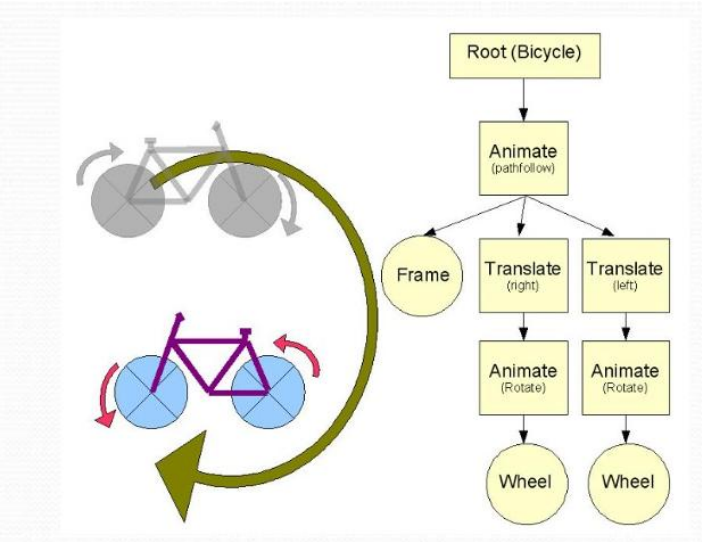


- A scene graph can contain animation nodes, which apply a transformation depending on the time. These can be used to animate elements of the scene.

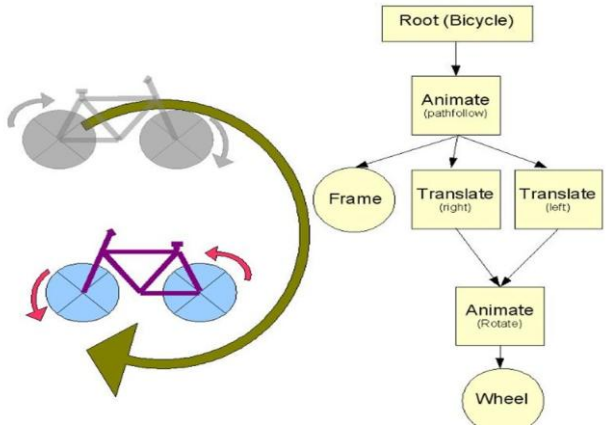


- The action of any node is applied to all of its children. If we insert a node that implements path following, we can cause the whole model to move along a

specified path. Like all computer animation, this movement is an illusion; the scene is drawn with slightly different positional parameters each frame.



- Finally, a scene graph may reuse a common component (subtree). In this example, we assume that both wheels will always be rotating at identical speeds.



Scene Graphs & Collision Detection

- Scene graphs can be used to speed up collision detection by incorporating a bounding volume at each node.
- A bounding volume is a defined region of space in which all the objects in the node's subtree reside. If an object does not intersect with a high-level bounding volume, it cannot intersect any object below that node in the hierarchy.
- The bounding volumes are calculated by performing a reverse traversal of the tree, merging the bounding volumes of children to create a bounding volume for the parent.
- A scene graph used this way is called a Bounding Volume Hierarchy (BVH). Some game engines keep the BVH and scene graph separate.

Modeling: The Camera

- Finally, we need a model of the virtual camera, which can be sophisticated, including:

    - Field of view
    - Depth of field
    - Distortion
    - Chromatic aberration
- **Interactive graphics (OpenGL)**:
    - Camera pose: position & orientation
    - Captured in viewing transform (i.e., modelview matrix)
    - Pinhole camera model
    - Field of view
    - Aspect ratio
    - Near & far clipping planes

Modeling: The Camera (contd)

- Camera parameters are encapsulated in a projection matrix (homogeneous coordinates - 4x4 matrix).
- The projection matrix premultiplies the viewing matrix, which premultiplies the modeling matrices.
- OpenGL uses viewing and modeling transforms into the model view matrix.

Rasterization

- Rasterization translates 3D shapes into 2D images by breaking the shapes down into triangles, mapping these triangles onto the pixel grid of the screen, and coloring each pixel as needed. This method is quick and efficient.
    - It directly converts shapes into pixels.
    - Modern GPUs are highly optimized for rasterization, leading to efficient rendering and making the most of the available hardware capabilities.
    - Rasterization benefits from widespread support in graphics APIs like DirectX and OpenGL.

Clipping

- Clipping in rasterization means to literally clip the data that need to be rendered to the area that makes up a screen.
- This has two main purposes:

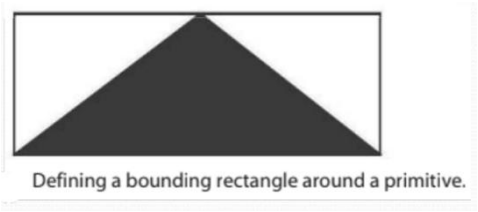
i. To handle the case when data outside of the viewport are out of range. (When using an array to render the canvas, you can receive an “out-of-bounds” logic error if the out-of-range data aren’t taken care of and you try to set elements beyond the array’s range.)

ii. To avoid processing data that are not within the viewport of the screen.

Scan-Line Conversion

- In rasterization, one of the processes is to define a bounding rectangle that encloses the pixel area range where a polygon is to be drawn.

  - Every row of pixels that are processed is known as a scan-line.
  - As each line is scanned, the pixels that make up the line are shaded as necessary.
  - When all of the rows of lines that make up the rectangular area have been processed, the primitive is considered drawn to the screen's canvas.


Defining a bounding rectangle around a primitive.

Pixel Overdraw

- In rasterization, there is no way to determine if a primitive has already been drawn in a location or if the existing primitive is closer or further than the primitive currently being rendered.
- This leads to what is known as pixel overdraw and can be a serious performance bottleneck problem in video games.
- Id Software had pixel overdraw problems during the development of Quake in the mid-1990s, which they solved using clever data structures and algorithms.

MODULE 5

Artificial Intelligence in Games for Move Prediction and Optimization

GAMES FOR ARTIFICIAL INTELLIGENCE

- Games offer an ***ideal domain*** for the study of artificial intelligence.
- The ***complexity*** and ***interestingness*** of games make them desirable for AI research.
- From a computational complexity perspective, many games are ***NP-hard*** (NP = nondeterministic polynomial time), meaning that the worst-case complexity of solving them is very high.
- Algorithms for solving such games can run for a very long time.
- Complexity varies based on the game's properties.
- Examples of NP-Hard games:
  - **Mastermind**
  - **Lemmings** (Psygnosis, 1991)

- **Minesweeper** (Microsoft)

WHY ARTIFICIAL INTELLIGENCE FOR GAMES?

- AI can improve games simply by playing them.
- AI plays games with two core objectives:
  - Play well (optimize performance)
  - Play believably (human-like or interestingly)
- AI can control:
  - Player characters (PCs)
  - Non-player characters (NPCs)
- AI that plays well as a **player character** is useful for:
  - Automatic game testing
  - Game design evaluation
- AI that plays well as a **non-player character** enables:
  - Dynamic difficulty adjustment
  - Automatic game balancing
  - Personalized player experiences

PANORAMIC VIEWS OF GAME AI

1. Methods (Computer) Perspective

- Focuses on AI methods used in games.
- **Evolutionary Computation**: Playing to win, generating content, modeling players, believable play.
- **Supervised Learning**: Dominant in player experience and behavioral modeling.
- **Behavior Authoring**: Primarily for gameplay control.
- **Reinforcement Learning and Unsupervised Learning**: Limited use; dominant only in playing to win and behavior modeling.
- **Tree Search**: Primarily for playing to win and computational narrative planning.

2. End User (Human) Perspective

- Emphasis on the end user (product/solution outcome).
- Three core dimensions:
  1. **Process AI follows** (Model or Generate)
  2. **Game context** (Content or Behavior)
  3. **End user** (Designer, Player, AI Researcher, Producer/Publisher)
- Example:
  - AI can model a player's affective state or generate a level.

3. Player – Game Interaction Perspective

- Emphasizes player experience and behavior.
- Player modeling focuses on interaction between player and game.
- Game content influenced by **procedural content generation**.
- NPC behavior informed by AI research on:
    - Winning strategies
    - Believability

WHAT IS "GAME AI"?

- "Game AI" refers to a broad set of algorithms from:
    - Control theory
    - Robotics
    - Computer graphics
    - Computer science
- Most video games include NPCs controlled by Game AI.
- In industry, Game AI often simply refers to the code controlling NPCs, regardless of complexity.

HISTORY OF GAME AI

- **Nim (1951)**: First application of Game AI by **Christopher Strachey**.
- **Chess (1952)**: Developed by **Dietrich Prinz**.

Examples of Games with Very Good AI

- **Blizzard Entertainment** titles
- **Far Cry 2** (Ubisoft Montreal)
- **Tom Clancy's Splinter Cell: Blacklist** (Ubisoft Montreal)

WHY USE AI TO PLAY GAMES?

- AI plays a very important role in the gaming industry.
- Most commonly used as the **opponent** in games.
- Four main uses:
    1. Playing to win in a player role
    2. Playing to win in a non-player role
    3. Playing for experience in the player role
    4. Playing for experience in a non-player role

1. Playing to Win in a Player Role

- AI used for **testing games** during development (simulation-based testing).
- Human-like playing is achievable.
- Games serve as excellent AI testbeds because of their human-like skill progression.
- Strategic games like **Chess**, **Checkers**, and **Go** require strong AI.
- In games with hidden information, sometimes AI cheats (accessing hidden state) to increase challenge.

2. Playing to Win in a Non-Player Role

- AI primarily controls **Non-Playable Characters (NPCs)**.
- NPCs are designed to be **entertaining** or **human-like**, not just challenging.
- Strategy games like **Civilization** and **XCOM: Enemy Unknown** need strong NPC AI.
- Racing games use AI for NPC cars that adapt to player speed to maintain challenge.

3. Playing for Experience in the Player Role

- AI is used for **human-like simulation**, not just winning.
- Helps automatically evaluate game content quality.
- Required for **demo modes** to show players how to play.
- AI must play like a human for accurate evaluation, but differences can arise based on the algorithm and game type.

4. Playing for Experience in a Non-Player Role

- AI often controls NPCs whose purpose is **not primarily to win**.
- NPCs serve multiple purposes:
    - Act as adversaries
    - Provide assistance
    - Guide players
    - Form part of puzzles
    - Drive narratives
    - Create emotional experiences
- NPC roles vary greatly, from side characters to main bosses.
- AI must create challenges that players can **learn and counter** over time.

WHICH GAMES CAN AI PLAY?

Board Games

- Easy to implement AI algorithms.
- Chess is commonly used for AI research.
- Focus on **adversarial planning** (planning against an opponent).
- Skill demands are narrow.
- Board games often have **simple discrete states** and **deterministic outcomes**.

Card Games

- Centered around one or more decks of cards.
- Most card games involve **hidden information**.
- Example: **Poker**.
- Requires **prediction**, **action**, and **reaction** strategies.

Classic Arcade Games

- Popular games from the 1970s and 1980s.
- Commonly used as AI benchmarks.
- Require **fast reactions** and **precise timing**.

Strategy Games

- Players control **multiple characters or units**.
- Objective: **Conquer or win a conflict**.
- Can be **turn-based** or **real-time**.
- Difficult for AI due to **large number of possible outcomes**.

Racing Games

- Players control vehicles or characters to **reach a goal quickly**.
- Involve **multi-tasking** and **high skill depth**.
- AI agents are trained on **different tracks and conditions**.
- Example: **Forza** (uses AI trained on human driving data).

First Person Shooter (FPS) Games

- Fast-paced games where **quick perception and reaction** are crucial.
- AI has **faster reaction times** than humans.
- Challenges for AI:
    - **Visual input** processing
    - **Orientation** and **movement** in a 3D environment
    - **Predicting enemy actions and locations**
    - **Team-based collaboration** in some modes

AI ALGORITHMS USED IN GAMES

- **Ad-hoc authoring** (custom-made logic)
- **Tree Search**
- **Evolutionary Computation**
- **Supervised Learning**
- **Reinforcement Learning**
- **Unsupervised Learning**

PATHFINDING ALGORITHMS

Algorithms that help AI find the best way from one point to another.

Common Pathfinding Algorithms:

1. **Breadth First Search (BFS)**
2. **Depth First Search (DFS)**
3. **Dijkstra's Algorithm**
4. **Greedy Search**
5. **A* (A Star) Algorithm**
6. **D***

Example Problem Statement

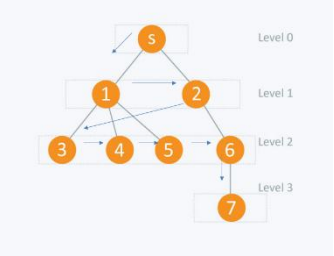
Story:

- Shizuka invites Nobita to have cake.
- Nobita is late because Gian made him play baseball.
- If Nobita delays more, Shizuka might invite Dekisuki instead.
- Doraemon (you) must **find the best path to Shizuka's house** quickly!

Solution:

- Use pathfinding algorithms like **BFS** or **DFS** to find the shortest/fastest route.

Breadth First Search (BFS)

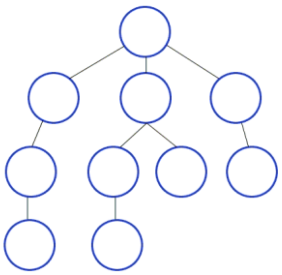


- Treat the neighborhood like **layers**.
- Explore **all nodes** at the current layer before moving to the next layer.
- **If destination is found**, stop – you found the shortest path!

Concepts:

- **Path**: Current explored path.
- **Visited**: Nodes already explored.
- **Tentative**: Nodes being considered for exploration.

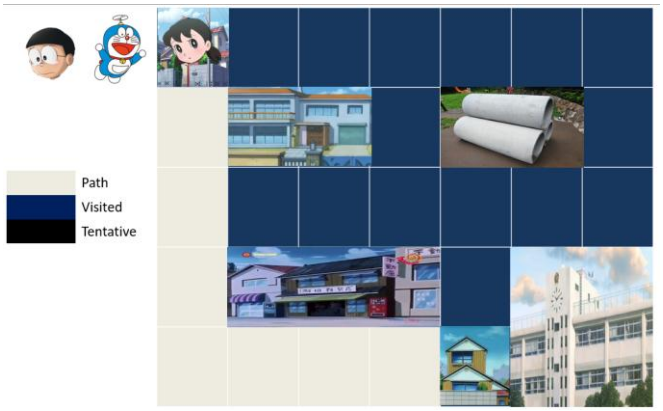
Depth First Search (DFS)



- Explore **as far as possible** along one path before **backtracking**.
- Continue this process until **all paths** are explored.

DFS in Trees:

- **Preorder**: Visit node, then children.
- **Postorder**: Visit children, then node.
- **Inorder** (for binary trees): Left subtree → Node → Right subtree.



DFS in Graphs:

- **Recursive Approach**:

```
procedure dfs(vertex v):
    visit(v)
    for each neighbor u of v:
        if u is undiscovered:
            dfs(u)
```

- **Iterative Approach**:
    - Use a **stack** instead of a queue.
    - Mark nodes as discovered **after popping** from the stack.
    - Use **reverse iterator** to match the recursive method's order.

Tic Tac Toe using DFS

Using recursion:

1. **Check if the game is over** (win or draw).
2. **Iterate** through all empty squares:
    - Make a move (X or O).
    - **Recursively** call the same function with the updated board.
    - Choose the move that gives the best outcome for the current player.

Dijkstra's Algorithm

Goal: Find the **shortest path** from a source node to all other nodes.

Steps:

1. **Initialize** all distances as infinity (∞), except the source (distance 0).
2. Create a **set** of nodes whose shortest distance is finalized.
3. While there are still nodes left:
    - Pick the node with the **smallest distance**.
    - Add it to the finalized set.
    - **Update distances** of its neighboring nodes if a shorter path is found.

Example: Dijkstra's Algorithm Steps

- Imagine points A, B, C, D.
- Initially, only A's distance is 0, others are ∞.
- Step by step, pick the nearest node and update distances.

Applications of Dijkstra's Algorithm

- **Google Maps** (for shortest route finding – although they use even more advanced algorithms now).
- **Network Routing Protocols** (like OSPF in networking).

Greedy Algorithm

- **Definition**: Chooses the best option at every step (locally optimal), hoping for a globally optimal result.
- **Pros**: Fast and simple.
- **Cons**: Often doesn't guarantee the best overall solution.
- **Example**: *Crystal Quest* game demo uses a greedy algorithm to collect crystals, which fails when obstacles are involved.

A* Algorithm

- **Definition**: Combines both **actual cost** from the start (g) and **heuristic estimated cost** to goal (h).
- **Formula**: f(n) = g(n) + h(n)
- **Why it's smart**: Chooses paths that balance efficiency and goal proximity.
- **Used in**: Many games where **speed** is crucial and **perfect accuracy** isn't needed.
- **Steps**:

```
1. Initialize the OPEN list (nodes to be evaluated)
2. Initialize the CLOSED list (nodes already evaluated)

3. Add the starting node to the OPEN list with:
   f = 0, g = 0, h = heuristic estimate to goal

4. While the OPEN list is not empty:
   a) Find the node `q` with the lowest f in the OPEN list
   b) Remove `q` from the OPEN list
   c) Generate q's 8 successors
   d) For each successor:
       i) If successor is the goal:
           - Stop the search
           - Set:
               successor.g = q.g + dist(q, successor)
               successor.h = heuristic to goal
               successor.f = successor.g + successor.h

      ii) If a node with the same position as successor is in the OPEN list
          with a lower f → skip this successor

     iii) If a node with the same position is in the CLOSED list
          with a lower f → skip this successor
          Else → add the successor to the OPEN list

   e) Add node `q` to the CLOSED list

End (while loop)
```

D* Algorithm

- **Definition**: Dynamic version of A*, adjusts to **changing terrain or goals** during execution.
- **Used in**:
    - **Games** with unknown or changing environments.
    - **Robotics**, like NASA Mars rovers and DARPA Urban Challenge.
- **Variants**:
    - **D***: Original.
    - **Focussed D***: More efficient version.
    - **D* Lite**: Simplified version used in real-time systems.

Minimax Algorithm

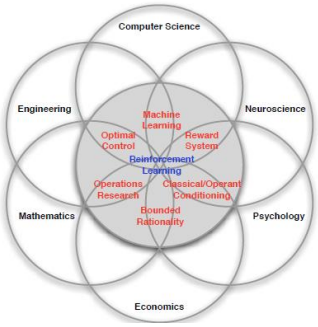
- **Used in**: Turn-based games like **Tic-Tac-Toe**.
- **Goal**: Maximize the minimum gain (minimize opponent's best outcome).
- **Steps**:
    1. Generate the game tree.
    2. Evaluate terminal states.
    3. Back-propagate values up the tree.
    4. Choose the move that maximizes the outcome.

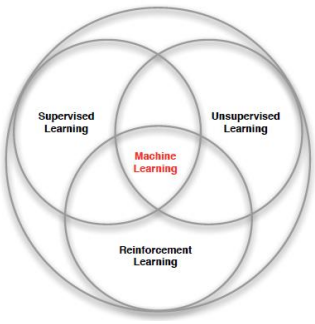
Alpha-Beta Pruning

- **Improves Minimax** by **cutting off branches** that don't influence the final decision.
- **Reduces computation time** without losing accuracy.
- **Key Terms**:
    - **Alpha**: Best option available to the maximizer so far.
    - **Beta**: Best option available to the minimizer so far.
- **When alpha ≥ beta**, prune the branch.

REINFORCEMENT LEARNING

- **Many Faces of Reinforcement Learning & Branches of Machine Learning**





🔸 Characteristics of Reinforcement Learning

What makes reinforcement learning different from other machine learning paradigms?

- There is no supervisor, only a **reward signal**
- Feedback is **delayed**, not instantaneous
- **Time matters** (sequential, non-i.i.d data)
- Agent's **actions affect** the subsequent data it receives

🔸 Examples of Reinforcement Learning

- Fly stunt manoeuvres in a helicopter
- Defeat the world champion at Backgammon
- Manage an investment portfolio
- Control a power station
- Make a humanoid robot walk
- Play many different Atari games better than humans

🔸 Rewards in Reinforcement Learning

- Based on the **reward hypothesis**
- All goals can be described by the **maximization of expected cumulative reward**

🔸 Examples of Rewards

Fly stunt manoeuvres in a helicopter

- +ve reward for following the desired trajectory
- −ve reward for crashing

Defeat the world champion at Backgammon

- +ve/−ve reward for winning/losing a game

Manage an investment portfolio

- +ve reward for each $ in the bank

Control a power station

- +ve reward for producing power
- −ve reward for exceeding safety thresholds

Make a humanoid robot walk

- +ve reward for forward motion
- −ve reward for falling over

Play Atari games better than humans

- +ve/−ve reward for increasing/decreasing score

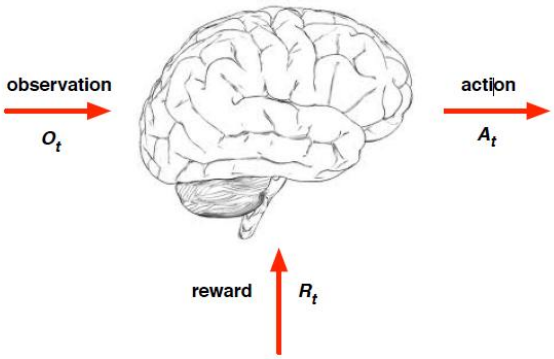
🔸 Sequential Decision Making

- Goal: select actions to **maximize total future reward**
- Actions may have **long-term consequences**
- Reward may be **delayed**
- Sometimes sacrificing immediate reward is better for **long-term gain**

Examples:

- A financial investment (may take months to mature)
- Refueling a helicopter (might prevent a crash hours later)
- Blocking opponent moves (might help win in later stages)

🔸 Agent and Environment



At each step t, the Agent:

- Executes action A_t
- Receives observation O_t
- Receives scalar reward R_t

The Environment:

- Receives action A_t
- Emits observation O_{t+1}
- Emits scalar reward R_{t+1}

🔎 *A* Search Algorithm*

🔸 A* Algorithm

The A* algorithm combines the uniform cost search and the Greedy search in the sense that it uses a priority (or cost) ordered queue (like uniform cost) and it uses an evaluation function (like Greedy) to determinate the ordering [5]. The evaluation function f is given by:

$$F(n) = h(n) + g(n)$$

where:

- h(n) is a heuristic function that estimates the cost from n to goal and
- g(n) is the cost so far to reach n.

Therefore, f(n) estimates total cost of path through n to goal.

The queue will be then sort based on estimates of full path costs (not just the cost so far, and not just the estimated remaining cost, but the two together).

It can be proven that if h(n) is admissible, then A* search will find an optimal solution.

A* is optimally efficient, i.e. there is no other optimal algorithm guaranteed to expand fewer nodes than A*. But it is not the answer to all path search problems as it still requires exponential time and space in general

Algorithm 3.2. A* search

```
Step 1. Let Q be a queue of partial paths (initially
root to root, length 0);
  Step 2. While Q is not empty or failure
       Step 2.1 if the first path P reaches the goal
       node then return success
       Step 2.2.remove path P from the queue;
       Step 2.3 extend P in all possible ways and add
       the new paths to the queue
       Step 2.4 sort the queue by the sum of two values:
       the real cost of P until now (g) and an estimate
       of the remaining distance (h);
       Step 2.5 prune the queue by leaving only the
       shortest path for each node reached so far;
Step 3. Return the shortest path (if success) or
failure.
```

🔁 *D* Algorithm*

D* Algorithm

- Named D* because it resembles A*, but is **dynamic**
- Arc costs can **change during traversal**
- If traversal is properly coupled to **replanning**, it guarantees **optimality**

D Lite*

- Used in goal-directed robot navigation in **unknown terrain**
- Robot observes which of its **8 adjacent cells are traversable**
- Robot starts at a **start cell** and moves to a **goal cell**
- Assumes unknown cells are traversable until proven otherwise
- Recomputes the shortest path **only when encountering obstacles**
- **Goal distances** of all traversable cells are calculated

- Shortest path is determined by **greedily decreasing goal distances**
- Only a **small number of goal distances** usually change, making the path update efficient

🧭 Navigation Mesh (NavMesh)

🔸 Overview

- An abstract data structure used in **AI pathfinding**
- Known since the **mid-1980s** in robotics (as meadow maps)
- Popular in **video game AI** since ~2000
- A navmesh consists of **2D convex polygons** marking traversable areas
- Adjacent polygons are connected via a **graph**
- **Pathfinding inside a polygon**: trivial (straight line)
- **Pathfinding between polygons**: use A* or similar
- **Avoids** expensive **collision detection checks**
- **Creation methods**:
    - Manually by level designers
    - Automatically from level geometry
    - Hybrid of both
- Typically assumes **static environments** (offline navmesh)
- Some research supports **dynamic updating** of navmeshes