SNS COLLEGE OF ENGINEERING

Kurumbapalayam (Po), Coimbatore – 641107

AN AUTONOMOUS INSTITUTION

Accredited by NBA-AICTE and Accredited by NAAC UGC with 'A' Grade

Approved by AICTE, New Delhi & Affiliates to Anna University, Chennai

ACADEMIC YEAR (2023-2024)

DEPARTMENT OF COMPUTER SCIENCE AND DESIGN

Course Code & Subject Name:19CD503 & Game Programming

III YEAR / V SEMESTER

UNIT – 4

3D GRAPHICS FOR GAME DESIGN

Genres of Games, Basics of 2D and 3D Graphics for Game Avatar, Game Components – 2D and 3D Transformations – Projections – Colour Models – Illumination and Shader Models – Animation – Controller Based Animation

GENRES OF GAMES

In game programming, different genres cater to various player preferences and experiences. These genres categorize games based on their gameplay mechanics, objectives, and overall experience.

1.Action: Fast-paced games involving challenges, reflexes, and hand-eye coordination. Sub-genres include platformers, beat 'em ups, shooters, and hack-and-slash games.

2.Adventure: Emphasizes exploration, puzzle-solving, and storytelling. Sub-genres include point-and-click adventures, narrative-driven games, and open-world exploration.

3.Role-Playing Games (RPG): Focuses on character development, storytelling, and decision-making. Sub-genres include action RPGs, turn-based RPGs, and massively multiplayer online RPGs (MMORPGs).

4.Strategy: Requires strategic thinking and planning. Sub-genres include real-time strategy (RTS), turn-based strategy (TBS), and 4X (explore, expand, exploit, exterminate) games.

5.Simulation: Simulates real-world activities or scenarios. Sub-genres include life simulations, vehicle simulations, and management simulations.

6.Puzzle: Challenges players with logical problems or puzzles. Sub-genres include physics-based puzzles, match-three games, and escape room puzzles.

7.Sports: Replicates real-world sports, such as football, basketball, racing, etc.

8.Horror: Aims to scare or unsettle the player through atmosphere, suspense, and often includes survival elements.

9.Fighting: Revolves around one-on-one combat between characters.

10.Racing: Involves racing vehicles or characters to reach a goal first.

11.Platformer: Involves navigating levels by jumping between platforms while avoiding obstacles

12.Survival: Challenges players to survive in a hostile environment by managing resources and dealing with threats.

13.Shooter: Focuses on combat using ranged weapons, such as first-person shooters (FPS) or third-person shooters (TPS).

14.Multiplayer Online Battle Arena (MOBA): Features teams of players battling in an arena setting.

15.Educational: Designed to teach or instruct players on various subjects or skills.

16.Idle or Incremental: Involves progress through minimal player interaction or by waiting over time.

These genres provide a framework for game developers to create diverse and engaging experiences for players, often combining elements from multiple genres to create hybrid or unique gaming experiences.

In game programming, creating avatars involves leveraging 2D or 3D graphics, each with its unique considerations:

BASICS OF 2D AND 3D GRAPHICS FOR GAME AVATAR:

2D Graphics for Game Avatars:

1.Sprites: In 2D games, characters or avatars are often represented as sprites. These are 2D images or animations that can be moved, flipped, or rotated to create the illusion of movement.

2.Animation: Characters are made up of multiple sprites arranged in a sequence to create fluid movements. This can involve walking, jumping, attacking, etc.

3.Pixel Art:2D avatars can be designed in pixel art, using a grid of pixels to create the characters. This style is common in indie games and retro-style games.

4.Character Design: Creating 2D avatars involves designing characters with attention to detail, personality, and expressiveness through art.

3D Graphics for Game Avatars:

1.3D Models: Avatars are built as three-dimensional models using polygons. They require modeling software like Blender, Maya, or 3ds Max to create the character's geometry.

2.Texturing: Textures are applied to 3D models to give them color, patterns, and details. These textures are created separately and then applied to the model.

3.Rigging and Animation :3D avatars are rigged, meaning a digital skeleton (armature) is placed within the model to enable movement. Animations are then created by manipulating the rig.

4.Character Design: Even in 3D, character design is crucial for avatars to have a distinct appearance and personality.

Common Practices for Both 2D and 3D:

1.Character Size and Proportions: Avatars need to be appropriately sized and proportioned relative to the game world and other elements.

2.Color and Style: Consistent art style and color palettes ensure avatars fit within the game's overall aesthetic.

3.Performance Optimization: Optimizing graphics is crucial, ensuring that the avatars don’t consume excessive resources, especially in real-time applications like games.

4.Integration with Game Mechanics: Avatars must be compatible with the game's mechanics, ensuring that they move and interact seamlessly within the game environment.

Creating avatars in games involves a mix of art, design, and technical skills. Understanding the basics of either 2D or 3D graphics and their implementation within the game is crucial for game developers to create engaging and visually appealing avatars.

Using Unity (C#) for 2D Graphics:

A basic example of how you might handle a 2D game avatar in Unity:

1.Sprite Creation: In Unity, you would import 2D sprites (character images) into the project. Unity provides an easy-to-use interface for creating and handling sprites.

2.Animator Component: Unity has an Animator component to manage sprite animations. You can create animation states and transitions between different animations like walking, jumping, and attacking.

```
// Example code for changing sprite animation state in Unity

Animator animator = Get Component<Animator>();
```

animator.SetBool("IsWalking", true); // Change animation state to walking

3.Character Movement: You can control the avatar's movement using C# scripts, utilizing Unity's Rigid body component for physics-based movement.

// Example code for moving the character in Unity

public float speed = 5f;

void Update() {

float horizontalInput = Input.GetAxis("Horizontal");

float verticalInput = Input.GetAxis("Vertical");

Vector3 movement = new Vector3(horizontal Input, vertical Input, 0) speed Time.deltaTime;

transform. Translate(movement);

}

Using Unity (C#) for 3D Graphics:

For 3D avatars, the principles are similar, but you work with 3D models, textures, rigging, and animations:

1.Import 3D Models: Import 3D models into Unity. Unity supports various 3D file formats.

2.Rigging and Animation: Rig the 3D model to create a skeleton for movement and create animations using Unity's Animation window or tools like Maya or Blender.

// Example code for playing an animation in Unity

Animator = Get Component<Animator>();

animator. Play ("Walk Animation");

3.Character Movement: Similarly, you control the avatar's movement using C# scripts, often involving the Unity Rigid body component for 3D physics.

// Example code for moving a 3D character in Unity

public float speed = 5f;

void Update () {

float horizontal Input = Input.GetAxis("Horizontal");

float vertical Input = Input.GetAxis("Vertical");

```
    Vector3 movement = new Vector3(horizontal Input, 0, vertical Input) speed
Time.deltaTime;
    transform. Translate(movement);
}
```

Unity simplifies many aspects of game development, providing a user-friendly interface and C# scripting for developing 2D and 3D games.

GAME COMPONENTS

In game programming, game components are modular building blocks that encapsulate specific functionalities or behaviors within a game. These components are used to create and manage various aspects of the game's mechanics, entities, interactions, and systems. They help organize code and facilitate reusability and maintainability in game development. Common game components include:

1.Transform Component: Handles the position, rotation, and scale of game objects within the game world.

2.Render Component: Manages the visual rendering of game objects, including sprites, models, textures, and shaders.

3.Collider Component: Deals with collision detection and response for game objects, enabling interactions between entities in the game.

4.Script/Behavior Component: Contains code that defines the behavior and functionality of game objects. It can control movement, AI, user input, and other interactive elements.

5.Audio Component: Manages sound effects, music, and other audio-related functionalities in the game.

6.Physics Component: Handles physical interactions, such as forces, gravity, and rigid body dynamics within the game environment.

7.UI Component: Manages the user interface elements, including menus, heads-up displays, and other on-screen information.

8.Animation Component: Controls the animations and transitions for characters, objects, and other game elements.

9.Particle System Component: Deals with creating and managing particle effects such as explosions, fire, smoke, etc.

10.Networking Component: Manages networking functionalities, enabling multiplayer, online play, or server-client interactions.

Game components are often used in conjunction with each other. For instance, a game object representing a character might contain components for rendering its visual representation, controlling its behavior, managing collisions, and handling its animations.

Designing a game with a modular and component-based approach allows for better code organization, reusability, and easier maintenance. Many game engines and frameworks provide built-in support for these components, simplifying the process of game development by providing pre-built components that developers can customize or extend to suit their specific game requirements.

2D AND 3D TRANSFORMATIONS:

Transformations in game programming involve manipulating the position, rotation, and scale of objects within the game world, facilitating the creation of dynamic and immersive environments. Both 2D and 3D games use different transformation techniques to place, orient, and resize objects. Here are the fundamental concepts for 2D and 3D transformations:

2D Transformations:

Translation (Position):

Description: Moving an object in 2D space along the x and y axes.

Formula: To move x units in the x-axis and y units in the y-axis:

- `newX = oldX + x`
- `newY = oldY + y`

Implementation in C# (Unity): Using the `Transform` component.

```
// Translate an object in Unity (C#)
transform.Translate(new Vector3(x, y, 0));
```

Rotation:

Description: Rotating an object around a specific point in 2D space.

Formula:

To rotate around a point:

`newX = centerX + (oldX - centerX) * cos(angle) - (oldY - centerY) * sin(angle)`

`newY = centerY + (oldX - centerX) * sin(angle) + (oldY - centerY) * cos(angle)`

Implementation in C# (Unity): Using the `Rotate` method of the `Transform` component.

// Rotate an object in Unity (C#)

transform.Rotate(new Vector3(0, 0, angle));

Scaling:

Description: Changing the size of an object in 2D space.

Formula: To scale by a factor:

- `newX = oldX * scaleFactorX`
- `newY = oldY * scaleFactorY`

Implementation in C# (Unity): Using the `localScale` property of the `Transform` component.

// Scale an object in Unity (C#)

transform.localScale = new Vector3(scaleFactorX, scaleFactorY, 1);

3D Transformations:

Translation (Position):

Description: Moving an object in 3D space along the x, y, and z axes.

Formula: To move x units in the x-axis, y units in the y-axis, and z units in the z-axis:

- `newX = oldX + x`
- `newY = oldY + y`
- `newZ = oldZ + z`

Implementation in C# (Unity): Using the `Transform` component.

// Translate an object in 3D space in Unity (C#)

transform.Translate(new Vector3(x, y, z));

Rotation:

Description: Rotating an object in 3D space around different axes (x, y, z).

Implementation in C# (Unity): Using the `Rotate` method of the `Transform` component.

// Rotate an object in 3D space in Unity (C#)

transform.Rotate(new Vector3(xAngle, yAngle, zAngle));

Scaling:

Description: Changing the size of an object in 3D space along the x, y, and z axes.

Implementation in C# (Unity): Using the `localScale` property of the `Transform` component.

// Scale an object in 3D space in Unity (C#)

transform.localScale = new Vector3(scaleFactorX, scaleFactorY, scaleFactorZ);

These transformations are essential in creating the visual and interactive aspects of 2D and 3D games, enabling the manipulation of game objects within the game world to bring life and dynamism to the gaming experience.

PROJECTIONS:

In game programming, projections refer to the methods used to transform three-dimensional (3D) scenes into two-dimensional (2D) representations that can be displayed on a screen. Projections are crucial for rendering 3D scenes in a way that accurately portrays depth, perspective, and realistic visual information. There are two primary types of projections:

1. Perspective Projection:

Description: Perspective projection simulates how humans perceive the world by representing depth and distance.

Characteristics:

Objects farther away appear smaller.

Lines converge at a vanishing point, following the principles of linear perspective.

Implementation:

In game programming, perspective projection is typically used in 3D game environments.

Algorithms like the perspective projection matrix are used to create the illusion of depth by adjusting the size and position of objects relative to the camera viewpoint.

Formula (in 3D rendering):

Transformation is done using a perspective projection matrix, which involves dividing the x, y, and z coordinates by the depth or distance from the camera's view.

2. Orthographic Projection:

Description: Orthographic projection portrays 3D scenes without simulating perspective.

Characteristics:

All lines are parallel and perpendicular to the view plane, resulting in no convergence of distant objects.

Objects maintain their size regardless of distance from the viewer.

Implementation:

Commonly used in 2D games and certain UI elements in 3D games.

Involves an orthographic projection matrix to project 3D scenes onto a 2D plane without depth consideration.

Formula (in 3D rendering):

Transformation is done using an orthographic projection matrix, which retains the original size and shape of objects without considering their distance from the camera.

Implementation in Game Programming:

Graphics APIs and Libraries: Game engines and graphics libraries (such as OpenGL, DirectX, Vulkan, and game development engines like Unity and Unreal Engine) provide methods to handle these projections.

Camera Setup: Game developers configure the camera's projection matrix or parameters to define the type of projection (perspective or orthographic).

Viewport Transformations: The projection is combined with other transformations like model, view, and viewport transformations to render the final image on the screen.

These projections are essential in game programming to provide players with a realistic, immersive, and visually accurate representation of the game world in both 2D and 3D environments. The choice of projection impacts the visual style and realism of the game, influencing how players perceive and interact with the game world.

COLOR MODELS:

Color models in game programming are essential for defining and representing colors in a digital environment. They provide a systematic way to express and manipulate colors, which are crucial for rendering visuals and creating immersive game experiences. Several color models are commonly used in game programming:

1. RGB (Red, Green, Blue):

Description: RGB is an additive color model in which various colors are produced by combining red, green, and blue light in varying intensities.

Usage in Games: RGB is extensively used in game development for defining colors in digital displays, where each pixel's color is composed of different levels of red, green, and blue light.

2. RGBA (Red, Green, Blue, Alpha):

Description: Similar to RGB, RGBA includes an additional channel (alpha) representing the pixel's transparency or opacity.

Usage in Games: Essential for specifying not only color but also the transparency level of objects or elements in the game environment, enabling effects like transparency, translucency, and fading.

3. CMYK (Cyan, Magenta, Yellow, Black):

Description: CMYK is a subtractive color model primarily used in printing. Cyan, magenta, and yellow are combined to create colors, while black is added to improve color depth and to print pure blacks.

Usage in Games: CMYK is less commonly used in games directly, as games are typically designed for digital screens rather than print. However, it might be used for certain design aspects, especially for assets intended for print media related to the game.

4. HSL (Hue, Saturation, Lightness):

Description: HSL is a cylindrical-coordinate representation of colors. Hue defines the type of color (e.g., red, blue), saturation represents the intensity of the color, and lightness determines the brightness of the color.

Usage in Games: Sometimes used for adjusting and manipulating colors in game design, such as in character customization or specific visual effects.

5. HSV (Hue, Saturation, Value):

Description: Similar to HSL but with 'value' instead of 'lightness'. Value is a measure of the brightness of a color.

Usage in Games: Similar to HSL, HSV is utilized for color manipulation in games, especially for applications requiring fine-tuning of color attributes or in creating varied visual effects.

6. YUV (Luminance, Chrominance):

Description: YUV is a color space separating luminance (brightness) and chrominance (color information). It's used in video encoding and transmission.

Usage in Games: Sometimes used in certain video game development scenarios, especially in video playback or encoding-related features.

These color models provide a structured way to represent and manipulate colors in game development, enabling developers to create vibrant visuals, implement various effects,

and ensure consistency in rendering across different devices and platforms. The choice of color model often depends on the specific requirements of the game and the intended visual outcome.

ILLUMINATION AND SHADER MODELS:

Illumination and shader models are integral aspects of game programming that contribute significantly to the visual aesthetics and realism of rendered graphics in video games. These elements influence how light interacts with objects in a scene, affecting the appearance of materials and surfaces.

Illumination Models:

Illumination models in game programming focus on simulating how light interacts with objects and their surfaces. These models help determine how light affects the color, brightness, and shading of objects in a 3D scene. Some common illumination models include:

1.Ambient Lighting: Represents the light that is scattered or bounced around in the environment, providing a base level of illumination to prevent scenes from appearing completely dark in areas not directly lit by a light source.

2.Diffuse Reflection: Describes how light scatters or spreads when it hits a surface, illuminating the surface uniformly. It's responsible for the primary color and brightness of an object.

3.Specular Reflection: Refers to the shiny or reflective highlights on surfaces when light is reflected off in a concentrated manner. This aspect is crucial for representing glossy or metallic surfaces.

4.Emissive Lighting: Represents surfaces that emit light themselves, such as glowing objects or light sources.

Shader Models:

Shaders are programs that describe the rendering process of an object in a scene, governing how the surfaces are lit, colored, and shaded. Shader models define the algorithms and operations for simulating various lighting effects and material properties. Some common shader models in game programming are:

1.Vertex Shaders: Operate on each vertex of a 3D model and are responsible for transformations like position, normal vector calculation, and passing data to the next stage.

2.Fragment (Pixel) Shaders: Handle individual pixels and are responsible for determining the final color of each pixel. They incorporate lighting calculations, textures, and effects to determine the pixel's appearance.

3.Geometry Shaders: Generate additional geometry or modify existing geometry on the GPU, allowing for effects such as tessellation, particle effects, or silhouette enhancements.

4.Tessellation Shaders: Control the level of detail for geometry, adjusting the complexity of 3D models dynamically.

Shader Effects:

Normal Mapping: Simulates finer surface details without adding extra geometry by altering the way light interacts with surfaces based on normal maps.

Parallax Mapping: Creates an illusion of depth by displacing texture coordinates based on a height map, giving the impression of surface relief.

Screen-Space Reflections (SSR): Simulates reflections based on what's visible on the screen, enhancing the realism of reflective surfaces.

Global Illumination (GI): Simulates indirect light bounces, enhancing the realism of lighting in the scene.

The choice and implementation of illumination and shader models in game development significantly impact the overall visual quality and realism of the game's graphics, contributing to immersive and visually appealing experiences for players.

ANIMATION:

Animation in game programming involves bringing characters, objects, and environments to life by creating the illusion of movement and action. It's a crucial aspect of game development that enhances the player's immersion and engagement. There are various types of animations used in games:

1.Keyframe Animation:

Description: Keyframe animation involves creating distinct poses or "keyframes" at specific points in time. The computer interpolates between these keyframes to create fluid motion.

Usage: Often used for character animations, object movements, and cutscenes.

2.Skeletal Animation:

Description: Involves using a hierarchical structure of bones (skeleton) and skinning (attaching the model to the skeleton) to animate characters.

Usage: Commonly used for character movements and complex animations such as walking, running, and combat.

3. Morph Target Animation (Blend Shapes):

Description: Uses different "morph targets" or shapes to interpolate between various mesh positions. This is often used for facial expressions, lip-syncing, and shape changes.

Usage: Primarily employed for facial animations and shape-shifting effects.

4.Procedural Animation:

Description: Involves generating animation through algorithms or physics simulations rather than relying on pre-made animations.

Usage: Useful for creating natural movements or behaviors, such as cloth simulation, particle effects, or dynamic environment interactions.

5.Inverse Kinematics (IK):

Description: Helps in solving the movement of end-effectors based on the desired position, which can be useful for precise control of limb movements.

Usage: Often used for character feet placement on uneven terrains or hand placements in interaction with objects.

Animation Workflow in Game Development:

1.Asset Creation: Artists design and create the 3D models, textures, and rig the characters for animation.

2.Animation Software: Animators use specialized software like Blender, Maya, or 3ds Max to create animations. These animations are then exported to formats compatible with the game engine.

3.Integration with Game Engine: Game developers import the animations into the game engine and implement them using scripts or built-in tools for controlling the playback and blending of animations.

4.Programming & Scripting: Programmers write code to control the playback of animations based on in-game events, player actions, or environmental interactions.

5.Blending and Transition: The game engine allows for smooth transitioning between animations, blending different motions, and controlling the timing of the animations.

Animation is a critical part of game development that contributes to the overall gaming experience, bringing characters and worlds to life and making the gameplay more immersive and engaging for the players.

CONTROLLER-BASED ANIMATION:

Controller-based animation in game programming involves using controllers or input devices to drive and control the animations of characters or objects within a game. It allows developers to create responsive and dynamic animations that react to user input in real-time, enhancing the overall player experience

1. Understanding Input Devices:

Controllers: Gamepads, keyboards, or any input devices used by players to interact with the game.

Inputs: Various actions or commands (e.g., button presses, analog stick movement) initiated by players using these devices.

2. Mapping Inputs to Animations:

Developers map specific input actions to corresponding animations:

Movement: Inputs from analog sticks or directional keys can trigger walking, running, or other locomotion animations.

Actions: Button presses or combinations may activate combat moves, jumps, crouching, or interactions with the game world.

3. State-Based Animation Control:

Games often implement a state-based system to control animations based on the current state of the character or object:

Finite State Machines (FSMs): Different states (e.g., idle, walking, running, attacking) are defined, and the corresponding animations are played based on the current state.

4. Blending and Transitions:

Smooth transitions between animations are crucial to avoid abrupt or jarring changes. Techniques such as animation blending and transition systems are used:

Animation Blending: Allows for combining multiple animations (e.g., blending a walking animation with an aiming animation for more fluid movement in a shooting game).

Transitions: Provide smooth shifts from one animation state to another to avoid sudden changes in character movements.

5. Scripting and Implementation:

Game developers write scripts or code to handle the logic for animation control:

Input Handling: Capturing input events and translating them into animation triggers.

Animation Control: Orchestrating the playback and transitions between animations based on the received inputs or game states.

6. Player Feedback and Responsiveness:

Responsive animations create a more immersive gaming experience:

Visual Feedback: Ensure that the animations respond promptly to player actions, providing clear feedback for their input.

Responsiveness: Make sure that the animations feel connected to the player's commands, ensuring the character or objects move in sync with the user's input.

Controller-based animation is a crucial component in modern game development, allowing for dynamic and engaging player experiences by giving direct control over character movements and actions based on user input. This system of animation control significantly contributes to the overall feel and responsiveness of the game.