Computer Graphics

What is Computer Graphics?

Definition:
Creating, manipulating, and rendering visual content using computers.

Key Concepts:

- 2D/3D Rendering: Drawing shapes, textures, and models.
- Transformations: Moving, rotating, scaling objects.
- Lighting/Shading: Simulating how light interacts.
- Rasterization: Converting math into pixels.

OpenGL

- A low-level API for rendering 2D/3D graphics.
- Directly controls the GPU.
- Cross-platform but complex (boilerplate code).

What is raylib?

- Simple and easy-to-use library for video game programming
- C-based (works perfectly with C++)
- Hardware accelerated with OpenGL
- Features:
  - 2D/3D support
  - Input handling
  - Texture/Sound loading
  - Font rendering
  - Physics system (via raylib-physac)

Basic raylib Setup (C++)

```cpp
#include "raylib.h"

int main() {
    InitWindow(800, 600, "raylib Demo");

    while (!WindowShouldClose()) {
        BeginDrawing();
            ClearBackground(RAYWHITE);
            DrawText("Hello World!", 190, 200, 20, LIGHTGRAY);
        EndDrawing();
    }

    CloseWindow();
    return 0;
}
```

The Basic Game Loop Structure

```
while (!WindowShouldClose()) // Runs until user closes window
{
    // 1. INPUT HANDLING
    ProcessInput();

    // 2. GAME LOGIC UPDATE
    UpdateGameState();

    // 3. RENDERING
    RenderFrame();
}
```

1. Input Handling

- **Polling**: Checks current input state every frame
- **Event Processing**: Handles keyboard, mouse, gamepad
- **State Storage**: Tracks pressed/released states

```
void ProcessInput()
{
    if (IsKeyDown(KEY_RIGHT)) player.x += speed;
    if (IsMouseButtonPressed(MOUSE_LEFT)) ShootBullet();
}
```

2. Game State Update

- **Delta Time**: Time between frames (critical for smooth movement)
- **Entity Updates**: Positions, collisions, AI
- **Game Rules**: Score, health, win/lose conditions

```
void UpdateGameState(float deltaTime)
{
    player.position.x += velocity.x * deltaTime;
    UpdateEnemies();
    CheckCollisions();
}
```

3. Rendering Pipeline

```
void RenderFrame()
{
    BeginDrawing();            // Start GPU drawing batch
        ClearBackground(RAYWHITE); // Clear last frame

        DrawPlayer();          // Draw game objects
        DrawEnemies();
        DrawUI();              // Overlay interface
    EndDrawing();              // Submit to GPU
}
```

What Are Design Patterns?

Design patterns are reusable solutions to common software design problems. They’re like blueprints for organizing code in a way that’s flexible, maintainable, and scalable.

They’re templates, not copy-paste solutions.

State Design Pattern

- Behavioral design pattern
- Allows object to change behavior when state changes
- Game State Management:
  - Menu State
  - Playing State
  - Pause State
  - Game Over State

Problem

```
void Hero::handleInput(Input input)
{
  if (input == PRESS_B)
  {
    yVelocity_ = JUMP_VELOCITY;
    setGraphics(IMAGE_JUMP);
  }
}
```

Problem

There’s nothing to prevent “air jumping”

```
void Hero::handleInput(Input input)
{
  if (input == PRESS_B)
  {
    if (!isJumping_)
    {
      isJumping_ = true;
      // Jump...
    }
  }
}
```

```
void Heroine::handleInput(Input input)
{
  if (input == PRESS_B)
  {
    // Jump if not jumping...
  }
  else if (input == PRESS_DOWN)
  {
    if (!isJumping_)
    {
      setGraphics(IMAGE_DUCK);
    }
  }
  else if (input == RELEASE_DOWN)
  {
    setGraphics(IMAGE_STAND);
  }
}
```

```
void Heroine::handleInput(Input input)
{
  if (input == PRESS_B)
  {
    if (!isJumping_ && !isDucking_)
    {
      // Jump...
    }
  }
  else if (input == PRESS_DOWN)
  {
    if (!isJumping_)
    {
      isDucking_ = true;
      setGraphics(IMAGE_DUCK);
    }
  }
  else if (input == RELEASE_DOWN)
  {
    if (isDucking_)
    {
      isDucking_ = false;
      setGraphics(IMAGE_STAND);
    }
  }
}
```

Solution

```
class HeroineState
{
public:
  virtual ~HeroState() {}
  virtual void handleInput(Hero& hero, Input input) {}
  virtual void update(Hero& hero) {}
};
```

Solution

```
class Hero
{
public:
  virtual void handleInput(Input input)
  {
    state_->handleInput(*this, input);
  }

  virtual void update()
  {
    state_->update(*this);
  }

  // Other methods...
private:
  HeroState* state_;
};
```

```
class DuckingState : public HeroState
{
public:
  DuckingState()
  : chargeTime_(0)
  {}

  virtual void handleInput(Hero& hero, Input input) {
    if (input == RELEASE_DOWN)
    {
      // Change to standing state...
      hero.setGraphics(IMAGE_STAND);
    }
  }

  virtual void update(Hero& hero) {
    chargeTime_++;
    if (chargeTime_ > MAX_CHARGE)
    {
      hero.superBomb();
    }
  }

private:
  int chargeTime_;
};
```

Benefits

- Clean separation of state-specific logic
- Easy to add new states
- Maintainable code structure
- State transitions are explicit
- Reusable state components

Resources

- raylib Official Website
- raylib Cheatsheet
- Game Programming Patterns
- State Pattern Reference