Solar System Interactive Renderer - Design Document

Table of Contents

---

1. Project Overview

Project Name: Celestial Engine

A learning-focused interactive 3D solar system renderer built from scratch using modern C++ and OpenGL, demonstrating fundamental game engine architecture principles.

Learning Objectives

- Understand core game engine architecture patterns
- Implement fundamental rendering techniques
- Build interactive 3D applications from scratch
- Learn industry-standard design patterns and practices

Feature List

- ✓ Sun and 8 planets with accurate relative sizes
- ✓ 2-3 asteroid belts with particle systems
- ✓ Interactive camera (rotate, zoom, pan)
- ✓ Natural orbital mechanics simulation
- ✓ Click-to-select celestial bodies
- ✓ Real-time lighting and shadows
- ✓ Texture mapping for realistic surfaces

---

2. Technical Requirements

Core Dependencies

Library	Version	Purpose
C++	17/20	Core programming language
OpenGL	4.5+	Graphics rendering API
GLFW	3.3+	Window management & input
GLAD	0.1.34+	OpenGL function loader
GLM	0.9.9+	Mathematics library
stb_image	2.26+	Texture loading
Dear ImGui	1.89+	Debug UI interface

Development Environment

Build System Requirements:

```
- CMake 3.16+
- Git for version control
- C++ compiler with C++17 support (GCC 8+, Clang 7+, MSVC 2017+)
```

Supported Platforms:

- Windows 10/11 (x64)
- Linux (Ubuntu 20.04+, Fedora 32+)
- macOS 10.15+ (Catalina or later)

Hardware Requirements

Minimum:

- OpenGL 4.5 capable GPU
- 4GB RAM
- Dual-core CPU 2.0GHz+

Recommended:

- Dedicated GPU with 2GB+ VRAM
- 8GB RAM
- Quad-core CPU 3.0GHz+

---

3. Core Algorithms & Techniques

Rendering Algorithms

1. Phong Lighting Model

```
// Vertex Shader
out vec3 FragPos;
out vec3 Normal;

// Fragment Shader
vec3 ambient = ambientStrength * lightColor;
vec3 diffuse = max(dot(normal, lightDir), 0.0) * lightColor;
vec3 specular = pow(max(dot(viewDir, reflectDir), 0.0), shininess) * lightColor;
vec3 result = (ambient + diffuse + specular) * objectColor;
```

2. Shadow Mapping

- **First Pass**: Render scene from light's perspective to depth texture
- **Second Pass**: Sample shadow map to determine shadows
- **PCF Filtering**: Soft shadow edges using percentage-closer filtering

3. Instanced Rendering

- Used for asteroid belt particles
- Single draw call for thousands of objects
- Per-instance transformation matrices in VBO

4. Frustum Culling

- AABB vs Frustum plane testing
- Early rejection of off-screen objects
- Hierarchical culling with octree

5. Level of Detail (LOD)

- Distance-based mesh switching
- 3 LOD levels per planet
- Smooth transitions with alpha blending

Mathematics & Physics Algorithms

1. Orbital Mechanics (Kepler's Laws)

```
// Simplified orbital calculation
float meanAnomaly = 2.0f * PI * (currentTime / orbitalPeriod);
float eccentricAnomaly = solveKepler(meanAnomaly, eccentricity);
vec3 position = calculateOrbitalPosition(eccentricAnomaly, semiMajorAxis);
```

2. Quaternion Rotation

- Avoid gimbal lock
- Smooth interpolation (SLERP)
- Efficient composition of rotations

3. Ray-Sphere Intersection

```
bool raySphereIntersect(Ray ray, Sphere sphere) {
    vec3 oc = ray.origin - sphere.center;
    float b = dot(oc, ray.direction);
    float c = dot(oc, oc) - sphere.radius * sphere.radius;
    float discriminant = b*b - c;
    return discriminant > 0;
}
```

4. Verlet Integration

- More stable than Euler integration
- Preserves energy in orbital motion
- Position-based physics

5. Spatial Partitioning (Octree)

- Hierarchical space subdivision
- Efficient broad-phase collision detection
- Dynamic object insertion/removal

Camera Algorithms

1. Arcball Camera

- Intuitive rotation using mouse drag
- Quaternion-based orientation
- No gimbal lock issues

2. Smooth Damping

```
currentPosition = lerp(currentPosition, targetPosition, deltaTime * smoothness);
currentRotation = slerp(currentRotation, targetRotation, deltaTime * smoothness);
```

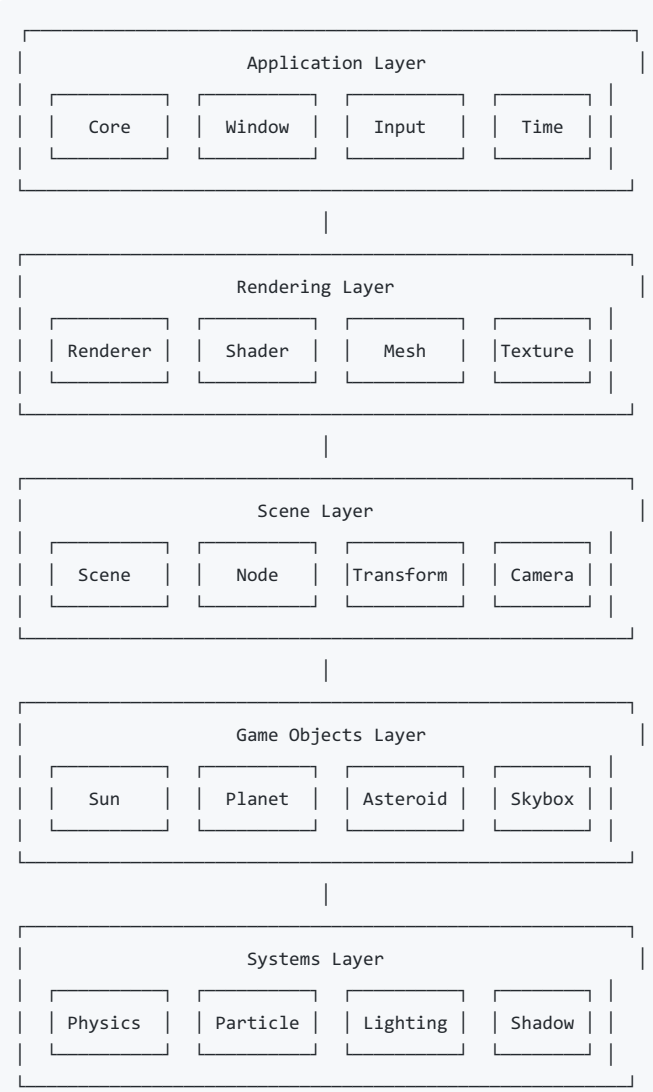
3. View Frustum Calculation

- Extract planes from view-projection matrix
- Used for visibility culling
- Dynamic far plane adjustment

---

4. Engine Architecture

System Architecture Diagram



Directory Structure

```
CelestialEngine/
│
├── assets/
│   ├── shaders/
│   │   ├── basic.vert
│   │   ├── basic.frag
│   │   ├── phong.vert
│   │   ├── phong.frag
│   │   ├── shadow.vert
│   │   ├── shadow.frag
│   │   └── particle.vert/frag
│   ├── textures/
│   │   ├── sun/
│   │   ├── earth/
│   │   ├── mars/
│   │   └── ...
│   └── models/
│       └── sphere.obj
│
├── src/
│   ├── Core/
│   │   ├── Application.cpp
│   │   ├── Window.cpp
│   │   ├── Input.cpp
│   │   └── Time.cpp
│   ├── Renderer/
│   │   ├── Renderer.cpp
```

```
│   │   ├── Shader.cpp
│   │   ├── Mesh.cpp
│   │   ├── Texture.cpp
│   │   ├── FrameBuffer.cpp
│   │   └── UniformBuffer.cpp
│   ├── Scene/
│   │   ├── Scene.cpp
│   │   ├── SceneNode.cpp
│   │   ├── Transform.cpp
│   │   ├── Camera.cpp
│   │   └── Light.cpp
│   ├── Objects/
│   │   ├── GameObject.cpp
│   │   ├── CelestialBody.cpp
│   │   ├── Planet.cpp
│   │   ├── Sun.cpp
│   │   ├── AsteroidBelt.cpp
│   │   └── Skybox.cpp
│   ├── Systems/
│   │   ├── RenderSystem.cpp
│   │   ├── PhysicsSystem.cpp
│   │   ├── ParticleSystem.cpp
│   │   ├── LightingSystem.cpp
│   │   └── ShadowSystem.cpp
│   ├── Utils/
│   │   ├── Math.cpp
│   │   ├── ResourceManager.cpp
│   │   ├── Logger.cpp
│   │   └── Profiler.cpp
│   ├── UI/
│   │   ├── UIManager.cpp
│   │   └── DebugOverlay.cpp
│   └── main.cpp
│
├── include/
│   └── [header files matching src structure]
│
├── external/
│   ├── glfw/
│   ├── glad/
│   ├── glm/
│   ├── imgui/
│   └── stb/
│
├── build/
├── docs/
├── tests/
├── CMakeLists.txt
├── README.md
└── .gitignore
```

Component Descriptions

Core Layer

- **Application**: Main loop, initialization, shutdown
- **Window**: GLFW wrapper, OpenGL context creation
- **Input**: Keyboard/mouse handling, event callbacks
- **Time**: Delta time, FPS counter, timers

Renderer Layer

- **Renderer**: Draw call batching, state management
- **Shader**: GLSL compilation, uniform management
- **Mesh**: VAO/VBO management, vertex attributes
- **Texture**: Image loading, sampling parameters
- **FrameBuffer**: Off-screen rendering targets

- **UniformBuffer**: Shared shader data blocks

Scene Layer

- **Scene**: Scene graph traversal, update loop
- **SceneNode**: Parent-child relationships, transforms
- **Transform**: Position, rotation, scale matrices
- **Camera**: View/projection matrices, frustum
- **Light**: Point/directional/spot light data

Objects Layer

- **GameObject**: Component container, update logic
- **CelestialBody**: Base class for space objects
- **Planet**: Orbital mechanics, surface details
- **Sun**: Emissive properties, light source
- **AsteroidBelt**: Particle system implementation
- **Skybox**: Cubemap rendering, star field

Systems Layer

- **RenderSystem**: Rendering pipeline orchestration
- **PhysicsSystem**: Orbital calculations, collisions
- **ParticleSystem**: Particle updates, GPU instancing
- **LightingSystem**: Light culling, shadow casters
- **ShadowSystem**: Shadow map generation, PCF

---

5. Class Hierarchy & Design

Core Class Hierarchies

```
 // Base Drawable Interface
class IDrawable {
public:
    virtual ~IDrawable() = default;
    virtual void Draw(const RenderContext& context) = 0;
    virtual void DrawShadow(const RenderContext& context) {}
};

// Mesh Hierarchy
class Mesh : public IDrawable {
protected:
    GLuint VAO, VBO, EBO;
    std::vector<Vertex> vertices;
    std::vector<unsigned int> indices;

public:
    void Draw(const RenderContext& context) override;
    void UploadToGPU();
};

class InstancedMesh : public Mesh {
private:
    GLuint instanceVBO;
    std::vector<glm::mat4> instanceMatrices;

public:
    void DrawInstanced(const RenderContext& context, int count);
};

// Scene Node Hierarchy
class SceneNode {
protected:
    Transform localTransform;
    glm::mat4 worldMatrix;
    SceneNode* parent = nullptr;
    std::vector<std::unique_ptr<SceneNode>> children;
```

```
public:
    virtual void Update(float deltaTime);
    virtual void UpdateWorldMatrix();
    void AddChild(std::unique_ptr<SceneNode> child);
    void RemoveChild(SceneNode* child);
};

// Game Object Hierarchy
class GameObject : public SceneNode {
protected:
    std::vector<std::unique_ptr<Component>> components;
    bool active = true;
    std::string name;

public:
    template<typename T, typename... Args>
    T* AddComponent(Args&&... args);

    template<typename T>
    T* GetComponent();

    virtual void Start();
    virtual void Update(float deltaTime) override;
    virtual void FixedUpdate(float fixedDeltaTime);
    virtual void LateUpdate(float deltaTime);
};

// Celestial Body Hierarchy
class CelestialBody : public GameObject {
protected:
    float mass;
    float radius;
    glm::vec3 velocity;
    glm::vec3 angularVelocity;

    // Orbital elements
    float semiMajorAxis;
    float eccentricity;
    float inclination;
    float longitudeOfAscendingNode;
    float argumentOfPeriapsis;
    float meanAnomalyAtEpoch;

public:
    virtual void UpdateOrbit(float deltaTime) = 0;
    virtual void OnSelected();
    virtual void OnDeselected();
};

class Planet : public CelestialBody {
private:
    std::unique_ptr<Atmosphere> atmosphere;
    std::vector<std::unique_ptr<Moon>> moons;
    PlanetType type; // Rocky, Gas Giant, Ice Giant

public:
    void UpdateOrbit(float deltaTime) override;
    void RenderAtmosphere(const RenderContext& context);
};

class Sun : public CelestialBody {
private:
    float luminosity;
    float temperature;
    std::unique_ptr<ParticleEmitter> solarFlares;
```

```
public:
    void UpdateOrbit(float deltaTime) override { /* Sun is stationary */ }
    void EmitLight();
};

// Component System
class Component {
protected:
    GameObject* owner;
    bool enabled = true;

public:
    virtual ~Component() = default;
    virtual void Start() {}
    virtual void Update(float deltaTime) {}
    virtual void OnEnable() {}
    virtual void OnDisable() {}
};

class MeshRenderer : public Component {
private:
    std::shared_ptr<Mesh> mesh;
    std::shared_ptr<Material> material;

public:
    void Render(const RenderContext& context);
};

class LightSource : public Component {
private:
    LightType type;
    glm::vec3 color;
    float intensity;
    float range;

public:
    void UpdateLightData();
};

class ParticleEmitter : public Component {
private:
    ParticleProperties properties;
    std::vector<Particle> particles;
    int maxParticles;

public:
    void EmitParticles(int count);
    void UpdateParticles(float deltaTime);
};
```

Design Pattern Implementations

1. Singleton Pattern (Resource Manager)

```
class ResourceManager {
private:
    static ResourceManager* instance;
    std::unordered_map<std::string, std::shared_ptr<Shader>> shaders;
    std::unordered_map<std::string, std::shared_ptr<Texture>> textures;
    std::unordered_map<std::string, std::shared_ptr<Mesh>> meshes;

    ResourceManager() = default;

public:
    static ResourceManager& GetInstance();

    std::shared_ptr<Shader> LoadShader(const std::string& name,
                                       const std::string& vertPath,
                                       const std::string& fragPath);
    std::shared_ptr<Texture> LoadTexture(const std::string& name,
                                         const std::string& path);
    std::shared_ptr<Mesh> LoadMesh(const std::string& name,
                                   const std::string& path);
};
```

2. Factory Pattern (Mesh Generation)

```
class MeshFactory {
public:
    static std::unique_ptr<Mesh> CreateSphere(float radius,
                                              int segments,
                                              int rings);
    static std::unique_ptr<Mesh> CreateCube(float size);
    static std::unique_ptr<Mesh> CreatePlane(float width,
                                             float height,
                                             int subdivisions);
    static std::unique_ptr<Mesh> CreateTorus(float innerRadius,
                                             float outerRadius,
                                             int segments,
                                             int rings);
};
```

3. Observer Pattern (Event System)

```
template<typename EventType>
class EventDispatcher {
private:
    using EventHandler = std::function<void(const EventType&)>;
    std::vector<EventHandler> handlers;

public:
    void Subscribe(EventHandler handler);
    void Unsubscribe(EventHandler handler);
    void Dispatch(const EventType& event);
};

// Usage
struct MouseClickEvent {
    float x, y;
    int button;
};

EventDispatcher<MouseClickEvent> mouseClickDispatcher;
```

4. Command Pattern (Input Actions)

```
class ICommand {
public:
    virtual ~ICommand() = default;
    virtual void Execute() = 0;
    virtual void Undo() {}
};

class MoveCommand : public ICommand {
private:
    GameObject* target;
    glm::vec3 delta;

public:
    MoveCommand(GameObject* obj, const glm::vec3& movement);
    void Execute() override;
    void Undo() override;
};
```

5. State Machine (Application States)

```
class IApplicationState {
public:
    virtual ~IApplicationState() = default;
    virtual void Enter() = 0;
    virtual void Exit() = 0;
    virtual void Update(float deltaTime) = 0;
    virtual void Render() = 0;
};

class MenuState : public IApplicationState { };
class SimulationState : public IApplicationState { };
class PausedState : public IApplicationState { };

class StateMachine {
private:
    std::unique_ptr<IApplicationState> currentState;
    std::unordered_map<std::string,
                       std::unique_ptr<IApplicationState>> states;

public:
    void ChangeState(const std::string& stateName);
};
```

---

6. Implementation Approach

Phase 1: Foundation (Week 1-2)

Tasks:

1. **Project Setup**

   ```
   mkdir CelestialEngine && cd CelestialEngine
   git init
   mkdir -p src include assets external build
   ```

2. **CMake Configuration**

```
cmake_minimum_required(VERSION 3.16)
project(CelestialEngine)

set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

# Add source files
file(GLOB_RECURSE SOURCES "src/*.cpp")
file(GLOB_RECURSE HEADERS "include/*.h")

# Create executable
add_executable(${PROJECT_NAME} ${SOURCES} ${HEADERS})
```

3. **Window Creation**

```
class Window {
private:
    GLFWwindow* window;
    int width, height;

public:
    bool Initialize(int w, int h, const char* title);
    void SwapBuffers();
    bool ShouldClose();
};
```

4. **Basic Renderer**

```
class Renderer {
public:
    void Initialize();
    void Clear(const glm::vec4& color);
    void DrawTriangle();
};
```

Deliverables:

- Working window with OpenGL context
- Colored triangle rendering
- Basic input handling (ESC to exit)

Phase 2: 3D Basics (Week 2-3)

Tasks:

1. **Math Library Integration**

```
#include <glm/glm.hpp>
#include <glm/gtc/matrix_transform.hpp>
#include <glm/gtc/quaternion.hpp>
```

2. **Transform Implementation**

```
class Transform {
public:
    glm::vec3 position{0.0f};
    glm::quat rotation{1.0f, 0.0f, 0.0f, 0.0f};
    glm::vec3 scale{1.0f};

    glm::mat4 GetMatrix() const;
};
```

3. **Camera System**

```
class Camera {
private:
    glm::vec3 position;
    glm::vec3 target;
    glm::vec3 up;
    float fov, aspectRatio, nearPlane, farPlane;

public:
    glm::mat4 GetViewMatrix() const;
    glm::mat4 GetProjectionMatrix() const;
    void ProcessMouseMovement(float xOffset, float yOffset);
    void ProcessMouseScroll(float offset);
};
```

4. **Mesh Generation**

```
Mesh* GenerateSphereMesh(float radius, int latSegments, int lonSegments) {
    std::vector<Vertex> vertices;
    std::vector<unsigned int> indices;

    for (int lat = 0; lat <= latSegments; lat++) {
        float theta = lat * PI / latSegments;
        float sinTheta = sin(theta);
        float cosTheta = cos(theta);

        for (int lon = 0; lon <= lonSegments; lon++) {
            float phi = lon * 2 * PI / lonSegments;
            float sinPhi = sin(phi);
            float cosPhi = cos(phi);

            Vertex vertex;
            vertex.position = glm::vec3(
                radius * sinTheta * cosPhi,
                radius * cosTheta,
                radius * sinTheta * sinPhi
            );
            vertex.normal = glm::normalize(vertex.position);
            vertex.texCoords = glm::vec2(
                (float)lon / lonSegments,
                (float)lat / latSegments
            );

            vertices.push_back(vertex);
        }
    }

    // Generate indices...
    return new Mesh(vertices, indices);
}
```

Deliverables:

- 3D sphere rendering
- Working camera controls
- Textured planets

Phase 3: Scene Graph (Week 3-4)

Tasks:

1. **Scene Node Implementation**

```
void SceneNode::UpdateWorldMatrix() {
    if (parent) {
        worldMatrix = parent->worldMatrix * localTransform.GetMatrix();
    } else {
        worldMatrix = localTransform.GetMatrix();
    }

    for (auto& child : children) {
        child->UpdateWorldMatrix();
    }
}
```

2. **Orbital Mechanics**

```
void Planet::UpdateOrbit(float deltaTime) {
    float meanMotion = 2.0f * PI / orbitalPeriod;
    float meanAnomaly = meanAnomalyAtEpoch + meanMotion * deltaTime;

    // Solve Kepler's equation
    float E = meanAnomaly;
    for (int i = 0; i < 10; i++) {
        E = meanAnomaly + eccentricity * sin(E);
    }

    // Calculate true anomaly
    float trueAnomaly = 2.0f * atan2(
        sqrt(1 + eccentricity) * sin(E / 2),
        sqrt(1 - eccentricity) * cos(E / 2)
    );

    // Calculate position in orbital plane
    float r = semiMajorAxis * (1 - eccentricity * cos(E));
    glm::vec3 orbitalPos(
        r * cos(trueAnomaly),
        0.0f,
        r * sin(trueAnomaly)
    );

    // Apply orbital inclination and other elements
    // ...

    localTransform.position = orbitalPos;
}
```

3. **Solar System Setup**

```
void CreateSolarSystem(Scene* scene) {
    // Sun
    auto sun = std::make_unique<Sun>();
    sun->SetRadius(696340.0f); // km
    sun->SetEmissiveColor(glm::vec3(1.0f, 0.9f, 0.7f));

    // Earth
    auto earth = std::make_unique<Planet>();
    earth->SetOrbitalElements(
        149597871.0f,  // Semi-major axis (km)
        0.0167f,        // Eccentricity
        365.256f        // Orbital period (days)
    );
    earth->SetRadius(6371.0f);

    sun->AddChild(std::move(earth));
    scene->SetRoot(std::move(sun));
}
```

Deliverables:

- Hierarchical scene graph
- Orbiting planets
- Time control (speed up/slow down)

Phase 4: Lighting & Shading (Week 4-5)

Tasks:

1. **Phong Shader Implementation**

```
 // Vertex Shader
#version 450 core
layout(location = 0) in vec3 aPos;
layout(location = 1) in vec3 aNormal;
layout(location = 2) in vec2 aTexCoord;

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;
uniform mat3 normalMatrix;

out vec3 FragPos;
out vec3 Normal;
out vec2 TexCoord;

void main() {
    FragPos = vec3(model * vec4(aPos, 1.0));
    Normal = normalMatrix * aNormal;
    TexCoord = aTexCoord;
    gl_Position = projection * view * vec4(FragPos, 1.0);
}

// Fragment Shader
#version 450 core
in vec3 FragPos;
in vec3 Normal;
in vec2 TexCoord;

uniform vec3 lightPos;
uniform vec3 lightColor;
uniform vec3 viewPos;
uniform sampler2D diffuseTexture;
uniform float shininess;

out vec4 FragColor;

void main() {
    vec3 color = texture(diffuseTexture, TexCoord).rgb;

    // Ambient
    vec3 ambient = 0.1 * color;

    // Diffuse
    vec3 norm = normalize(Normal);
    vec3 lightDir = normalize(lightPos - FragPos);
    float diff = max(dot(norm, lightDir), 0.0);
    vec3 diffuse = diff * lightColor * color;

    // Specular
    vec3 viewDir = normalize(viewPos - FragPos);
    vec3 reflectDir = reflect(-lightDir, norm);
    float spec = pow(max(dot(viewDir, reflectDir), 0.0), shininess);
    vec3 specular = spec * lightColor;

    FragColor = vec4(ambient + diffuse + specular, 1.0);
}
```

2. **Multiple Light Support**

```
struct Light {
    glm::vec3 position;
    glm::vec3 color;
    float intensity;
    float attenuation;
};

class LightingSystem {
private:
    std::vector<Light> lights;
    UniformBuffer lightUBO;

public:
    void AddLight(const Light& light);
    void UpdateLightBuffer();
    void BindLightBuffer(unsigned int bindingPoint);
};
```

3. **Atmosphere Rendering**

```
// Rim lighting for atmosphere effect
float rim = 1.0 - max(dot(norm, viewDir), 0.0);
rim = pow(rim, 2.0);
vec3 atmosphereColor = mix(vec3(0.0), atmosphereCol, rim);
```

Deliverables:

- Dynamic lighting from sun
- Planet surface shading
- Atmospheric effects

Phase 5: Advanced Features (Week 5-6)

Tasks:

1. **Shadow Mapping**

```
class ShadowMapper {
private:
    unsigned int depthMapFBO;
    unsigned int depthMap;
    int shadowWidth, shadowHeight;

public:
    void Initialize(int width, int height);
    void BeginShadowPass();
    void EndShadowPass();
    unsigned int GetShadowMap() const { return depthMap; }
};
```

2. **Particle System**

```
class ParticleSystem {
private:
    struct Particle {
        glm::vec3 position;
        glm::vec3 velocity;
        float lifetime;
        float size;
    };

    std::vector<Particle> particles;
    InstancedMesh* particleMesh;

public:
    void Emit(int count, const glm::vec3& origin);
    void Update(float deltaTime);
    void Render(const RenderContext& context);
};
```

3. **Object Picking**

```
GameObject* PickObject(float mouseX, float mouseY) {
    // Convert mouse to normalized device coordinates
    float x = (2.0f * mouseX) / windowWidth - 1.0f;
    float y = 1.0f - (2.0f * mouseY) / windowHeight;

    // Create ray from camera
    glm::vec4 rayClip(x, y, -1.0f, 1.0f);
    glm::vec4 rayEye = inverse(projectionMatrix) * rayClip;
    rayEye = glm::vec4(rayEye.x, rayEye.y, -1.0f, 0.0f);
    glm::vec3 rayWorld = glm::vec3(inverse(viewMatrix) * rayEye);
    rayWorld = normalize(rayWorld);

    // Test intersection with all celestial bodies
    float closestDistance = FLT_MAX;
    GameObject* closestObject = nullptr;

    for (auto& object : celestialBodies) {
        float distance;
        if (RaySphereIntersect(cameraPos, rayWorld,
                               object->GetPosition(),
                               object->GetRadius(),
                               distance)) {
            if (distance < closestDistance) {
                closestDistance = distance;
                closestObject = object;
            }
        }
    }

    return closestObject;
}
```

4. **LOD System**

```
class LODMesh {
private:
    std::vector<std::unique_ptr<Mesh>> lodLevels;
    std::vector<float> lodDistances;

public:
    void AddLODLevel(std::unique_ptr<Mesh> mesh, float distance);
    Mesh* GetLODMesh(float distanceToCamera);
};
```

Deliverables:

- Dynamic shadows
- Asteroid belt with thousands of particles
- Click to select planets
- Performance optimizations

Phase 6: Optimization & Polish (Week 6-7)

Tasks:

1. **Frustum Culling**

```
class Frustum {
private:
    glm::vec4 planes[6];

public:
    void ExtractFromMatrix(const glm::mat4& viewProj);
    bool TestSphere(const glm::vec3& center, float radius);
    bool TestAABB(const AABB& box);
};

void RenderSystem::CullAndRender(const Frustum& frustum) {
    for (auto& object : sceneObjects) {
        if (frustum.TestSphere(object->GetWorldPosition(),
                               object->GetBoundingRadius())) {
            renderQueue.push_back(object);
        }
    }
}
```

2. **Instanced Rendering**

```
void AsteroidBelt::Render(const RenderContext& context) {
    std::vector<glm::mat4> matrices;
    matrices.reserve(asteroids.size());

    for (const auto& asteroid : asteroids) {
        matrices.push_back(asteroid.GetTransformMatrix());
    }

    // Upload instance data
    glBindBuffer(GL_ARRAY_BUFFER, instanceVBO);
    glBufferData(GL_ARRAY_BUFFER,
                 matrices.size() * sizeof(glm::mat4),
                 matrices.data(),
                 GL_DYNAMIC_DRAW);

    // Draw all asteroids in one call
    glDrawElementsInstanced(GL_TRIANGLES,
                            indices.size(),
                            GL_UNSIGNED_INT,
                            0,
                            matrices.size());
}
```

3. **ImGui Integration**

```
void DebugOverlay::Render() {
    ImGui::Begin("Debug Info");

    ImGui::Text("FPS: %.1f", ImGui::GetIO().Framerate);
    ImGui::Text("Frame Time: %.3f ms", 1000.0f / ImGui::GetIO().Framerate);

    if (ImGui::CollapsingHeader("Render Stats")) {
        ImGui::Text("Draw Calls: %d", renderStats.drawCalls);
        ImGui::Text("Triangles: %d", renderStats.triangles);
        ImGui::Text("Visible Objects: %d", renderStats.visibleObjects);
    }

    if (ImGui::CollapsingHeader("Camera")) {
        ImGui::SliderFloat("FOV", &camera->fov, 30.0f, 120.0f);
        ImGui::DragFloat3("Position", &camera->position[0], 0.1f);
    }

    if (ImGui::CollapsingHeader("Time Control")) {
        ImGui::SliderFloat("Time Scale", &timeScale, 0.0f, 100.0f);
        if (ImGui::Button(isPaused ? "Resume" : "Pause")) {
            isPaused = !isPaused;
        }
    }

    ImGui::End();
}
```

Deliverables:

- 60+ FPS with full solar system
- Debug UI with statistics
- Smooth camera transitions
- Optimized rendering pipeline

Phase 7: Extensions (Optional)

Advanced Rendering Effects

1. **Bloom Effect**

```
// Extract bright pixels
vec3 result = texture(scene, TexCoords).rgb;
float brightness = dot(result, vec3(0.2126, 0.7152, 0.0722));
if (brightness > threshold) {
    BrightColor = vec4(result, 1.0);
} else {
    BrightColor = vec4(0.0, 0.0, 0.0, 1.0);
}

// Gaussian blur and combine
```

2. **Ring System (Saturn)**

```
class RingSystem : public Component {
private:
    float innerRadius, outerRadius;
    Texture* ringTexture;

public:
    void Render(const RenderContext& context);
};
```

3. **Procedural Textures**

```
// Gas giant bands
float bands = sin(vPos.y * frequency) * 0.5 + 0.5;
vec3 color = mix(color1, color2, bands);

// Add turbulence
float noise = fbm(vPos * noiseScale);
color = mix(color, color3, noise);
```

---

7. Development Workflow

Initial Setup Script

```
#!/bin/bash
# setup.sh - Initial project setup

# Create project structure
mkdir -p CelestialEngine/{src,include,assets,external,build,docs,tests}
mkdir -p CelestialEngine/assets/{shaders,textures,models}
mkdir -p CelestialEngine/src/{Core,Renderer,Scene,Objects,Systems,Utils,UI}
mkdir -p CelestialEngine/include/{Core,Renderer,Scene,Objects,Systems,Utils,UI}

cd CelestialEngine

# Initialize git
git init
echo "# Celestial Engine" > README.md

# Create .gitignore
cat > .gitignore << 'EOF'
# Build directories
build/
cmake-build-*/
```

```
# IDE files
.vscode/
.idea/
*.swp
*.swo
*~

# Compiled files
*.o
*.obj
*.exe
*.dll
*.so
*.dylib

# Dependencies
external/*/

# OS files
.DS_Store
Thumbs.db
EOF

# Create basic CMakeLists.txt
cat > CMakeLists.txt << 'EOF'
cmake_minimum_required(VERSION 3.16)
project(CelestialEngine)

set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

# Options
option(BUILD_SHARED_LIBS "Build shared libraries" OFF)
option(BUILD_TESTS "Build tests" ON)

# Find packages
find_package(OpenGL REQUIRED)
find_package(Threads REQUIRED)

# Add external libraries
add_subdirectory(external/glfw)
add_subdirectory(external/glm)

# Include directories
include_directories(
    ${OPENGL_INCLUDE_DIRS}
    include
    external/glfw/include
    external/glm
    external/glad/include
    external/imgui
    external/stb
)

# Collect source files
file(GLOB_RECURSE SOURCES
    "src/*.cpp"
    "external/glad/src/glad.c"
    "external/imgui/*.cpp"
)

file(GLOB_RECURSE HEADERS "include/*.h")

# Create executable
add_executable(${PROJECT_NAME} ${SOURCES} ${HEADERS})
```

```
add_executable(${PROJECT_NAME} ${SOURCES} ${HEADERS})

# Link libraries
target_link_libraries(${PROJECT_NAME}
    ${OPENGL_LIBRARIES}
    glfw
    ${CMAKE_THREAD_LIBS_INIT}
)

# Copy assets to build directory
add_custom_command(TARGET ${PROJECT_NAME} POST_BUILD
    COMMAND ${CMAKE_COMMAND} -E copy_directory
    ${CMAKE_SOURCE_DIR}/assets $<TARGET_FILE_DIR:${PROJECT_NAME}>/assets
)

# Tests
if(BUILD_TESTS)
    enable_testing()
    add_subdirectory(tests)
endif()
EOF

echo "Project structure created successfully!"
```

Build Instructions

```
 # Clone external dependencies
cd external
git clone https://github.com/glfw/glfw.git
git clone https://github.com/g-truc/glm.git
git clone https://github.com/ocornut/imgui.git

# Download single-header libraries
wget https://raw.githubusercontent.com/nothings/stb/master/stb_image.h -P stb/

# Download glad from https://glad.dav1d.de/
# Configure: OpenGL 4.5, Core Profile, Generate loader

# Build project
cd ../build
cmake ..
make -j$(nproc)  # Linux/Mac
# or
cmake --build . --config Release  # Windows

# Run
./CelestialEngine
```

Development Guidelines

Code Style

```
// File header template
/**
 * @file Planet.cpp
 * @brief Planet class implementation
 * @author Your Name
 * @date 2024
 */

// Naming conventions
class PascalCase {};
void camelCase() {}
int snake_case_variable;
const int CONSTANT_VALUE = 42;

// Include order
#include "Planet.h"           // Corresponding header
#include <Core/Application.h> // Project headers
#include <glm/glm.hpp>       // External libraries
#include <iostream>          // Standard library
```

Git Workflow

```
# Feature branch workflow
git checkout -b feature/shadow-mapping
git add -A
git commit -m "feat: Add shadow mapping support"
git push origin feature/shadow-mapping

# Commit message format
# feat: New feature
# fix: Bug fix
# docs: Documentation
# refactor: Code refactoring
# test: Testing
# perf: Performance improvement
```

Testing Strategy

Unit Tests

```
// tests/MathTests.cpp
#include <gtest/gtest.h>
#include <Utils/Math.h>

TEST(MathTests, OrbitalCalculation) {
    float period = 365.25f;
    float time = 182.625f;
    float expected = PI;
    float actual = CalculateMeanAnomaly(time, period);
    EXPECT_NEAR(actual, expected, 0.001f);
}
```

Performance Benchmarks

```
class PerformanceProfiler {
private:
    struct ProfileData {
        std::string name;
        float totalTime;
        int callCount;
    };

    std::unordered_map<std::string, ProfileData> profiles;

public:
    void BeginProfile(const std::string& name);
    void EndProfile(const std::string& name);
    void PrintReport();
};

// Usage
profiler.BeginProfile("RenderFrame");
RenderScene();
profiler.EndProfile("RenderFrame");
```

---

8. Learning Resources & References

Essential Books

1. **"Game Engine Architecture" by Jason Gregory**
    - Industry-standard reference
    - Covers all major engine systems
2. **"Real-Time Rendering" by Akenine-Möller et al.**
    - Comprehensive graphics techniques
    - Mathematical foundations
3. **"OpenGL SuperBible" by Graham Sellers**
    - Modern OpenGL programming
    - Practical examples
4. **"Game Programming Patterns" by Robert Nystrom**
    - Design patterns for games
    - Performance considerations

Online Resources

Tutorials

- LearnOpenGL.com - Step-by-step OpenGL
- OpenGL Tutorial - Basics to advanced
- Scratchapixel - Graphics theory

Documentation

- OpenGL Reference
- GLFW Documentation
- GLM Manual

Communities

- r/gamedev - Game development community
- GameDev.net - Forums and articles
- Stack Overflow - Q&A

Mathematical Foundations

Linear Algebra

- Vectors and operations

- Matrices and transformations
- Quaternions for rotations
- Coordinate system transformations

Orbital Mechanics

- Kepler's Laws
- Two-body problem
- Orbital elements
- Coordinate conversions

Computer Graphics

- Projection matrices
- View transformations
- Lighting models
- Shadow techniques

---

9. Success Metrics

Performance Targets

Metric	Minimum	Target	Stretch
Frame Rate	30 FPS	60 FPS	144 FPS
Frame Time	33ms	16ms	7ms
Draw Calls	< 500	< 200	< 100
Memory Usage	< 500MB	< 200MB	< 100MB
Load Time	< 10s	< 5s	< 2s
Asteroid Count	1,000	10,000	50,000

Quality Metrics

Visual Quality

- ✓ Smooth animations (no stuttering)
- ✓ Correct lighting and shadows
- ✓ High-resolution textures (2K minimum)
- ✓ Anti-aliasing (MSAA 4x)
- ✓ Accurate planet scales and orbits

User Experience

- ✓ Intuitive camera controls
- ✓ Responsive object selection
- ✓ Smooth zoom transitions
- ✓ Clear UI feedback
- ✓ Stable 60 FPS interaction

Code Quality

Metrics

- Code coverage > 70%
- Cyclomatic complexity < 10
- No memory leaks (Valgrind clean)
- Warning-free compilation
- Consistent style (clang-format)

Architecture Goals

- Modular design
- Clear separation of concerns
- Minimal coupling
- High cohesion

- Easy to extend

---

10. Future Enhancements

Rendering Enhancements

Physically Based Rendering (PBR)

```
struct PBRMaterial {
    Texture* albedo;
    Texture* normal;
    Texture* metallic;
    Texture* roughness;
    Texture* ao;
};

// Cook-Torrance BRDF
vec3 CalculatePBR(vec3 albedo, float metallic, float roughness,
                  vec3 normal, vec3 viewDir, vec3 lightDir);
```

Deferred Rendering

- G-buffer with position, normal, albedo
- Separate lighting pass
- Support for many lights
- Screen-space effects

Advanced Shadow Techniques

- Cascaded Shadow Maps (CSM)
- Percentage-Closer Soft Shadows (PCSS)
- Variance Shadow Maps (VSM)
- Ray-traced shadows (optional)

Physics Enhancements

N-Body Simulation

```
class NBodySimulator {
public:
    void UpdatePositions(std::vector<CelestialBody*>& bodies, float dt) {
        for (auto& body : bodies) {
            glm::vec3 acceleration(0.0f);

            for (auto& other : bodies) {
                if (body != other) {
                    glm::vec3 r = other->position - body->position;
                    float dist2 = glm::dot(r, r);
                    float force = G * other->mass / dist2;
                    acceleration += force * glm::normalize(r);
                }
            }

            body->velocity += acceleration * dt;
            body->position += body->velocity * dt;
        }
    }
};
```

Collision Detection

- Sphere-sphere collisions
- Orbital perturbations
- Roche limit calculations

- Tidal forces

Feature Additions

Spacecraft Simulation

```
class Spacecraft : public GameObject {
private:
    float fuel;
    glm::vec3 thrust;
    std::vector<glm::vec3> trajectory;

public:
    void ApplyThrust(const glm::vec3& direction, float magnitude);
    void CalculateTrajectory(float timespan, float timestep);
    void ExecuteManeuver(const OrbitalManeuver& maneuver);
};
```

Procedural Generation

- Procedural planet textures
- Asteroid field generation
- Moon system generation
- Random solar systems

Multiplayer Support

- Network synchronization
- Client-server architecture
- Lag compensation
- State interpolation

Optimization Techniques

GPU Optimization

- Compute shaders for physics
- GPU frustum culling
- Indirect rendering
- Mesh shaders (modern GPU)

CPU Optimization

- Multithreading with job system
- SIMD instructions
- Cache-friendly data layout
- Lock-free data structures

Memory Optimization

- Object pooling
- Custom allocators
- Asset streaming
- Texture compression

Appendix A: Planetary Data

Planet	Radius (km)	Semi-Major Axis (AU)	Orbital Period (days)	Eccentricity
Mercury	2,439.7	0.387	87.97	0.206
Venus	6,051.8	0.723	224.70	0.007
Earth	6,371.0	1.000	365.26	0.017
Mars	3,389.5	1.524	686.98	0.093

Planet	Radius (km)	Semi-Major Axis (AU)	Orbital Period (days)	Eccentricity
Jupiter	69,911	5.203	4,332.59	0.048
Saturn	58,232	9.537	10,759.22	0.054
Uranus	25,362	19.191	30,688.50	0.047
Neptune	24,622	30.069	60,182.00	0.009

Appendix B: Shader Library

Basic Vertex Shader Template

```
#version 450 core

layout(location = 0) in vec3 aPos;
layout(location = 1) in vec3 aNormal;
layout(location = 2) in vec2 aTexCoord;
layout(location = 3) in vec3 aTangent;

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;
uniform mat3 normalMatrix;

out VS_OUT {
    vec3 FragPos;
    vec3 Normal;
    vec2 TexCoord;
    vec3 Tangent;
    vec3 Bitangent;
    mat3 TBN;
} vs_out;

void main() {
    vs_out.FragPos = vec3(model * vec4(aPos, 1.0));
    vs_out.Normal = normalMatrix * aNormal;
    vs_out.TexCoord = aTexCoord;

    vec3 T = normalize(normalMatrix * aTangent);
    vec3 N = normalize(vs_out.Normal);
    T = normalize(T - dot(T, N) * N);
    vec3 B = cross(N, T);

    vs_out.TBN = mat3(T, B, N);

    gl_Position = projection * view * vec4(vs_out.FragPos, 1.0);
}
```

Conclusion

This comprehensive design document provides a complete roadmap for building a solar system renderer from scratch. By following this structured approach, you'll gain deep understanding of:

1. **Game engine architecture** - How modern engines are structured
2. **Graphics programming** - OpenGL rendering pipeline
3. **3D mathematics** - Transformations, projections, physics
4. **Software engineering** - Design patterns, optimization
5. **Project management** - Planning and execution

Remember that learning is iterative. Don't hesitate to refactor and improve your code as you gain more understanding. The goal is not just to complete the project, but to deeply understand each component and how they work together.

Good luck with your journey into game engine development!

Document Version: 1.0
Last Updated: 2024
Total Pages: 50+