Vulkan Scene Graph (VSG) Developer Guide

This guide provides a comprehensive overview of the Vulkan Scene Graph (VSG) library, compiled from the examples in this repository. It serves as a reference for developers creating new VSG applications.

Table of Contents

1. Introduction
2. Core Concepts
3. Memory Management
4. Scene Graph
5. Rendering
6. State Management
7. Animation
8. Stereo Rendering
9. Input and UI
10. Threading
11. Advanced Topics

Introduction

Vulkan Scene Graph (VSG) is a modern, high-performance scene graph library built on top of Vulkan. It provides:

- Type-safe, efficient C++ API
- Built-in memory management with smart pointers
- Visitor pattern for scene traversal
- Multi-threaded rendering support
- Efficient state management
- Cross-platform support

Core Concepts

Smart Pointers and Memory Management

VSG uses an intrusive reference counting system with `vsg::ref_ptr<T>` as its primary smart pointer. This design choice offers significant performance benefits over `std::shared_ptr`:

vsg::ref_ptr

```
// Creating objects - always use create() factory method
vsg::ref_ptr<vsg::Group> group = vsg::Group::create();
vsg::ref_ptr<vsg::QuadGroup> quadGroup = vsg::QuadGroup::create();

// ref_ptr is only 8 bytes (just a pointer)
// std::shared_ptr is 16 bytes (pointer + control block pointer)
```

Key characteristics: - **Intrusive reference counting**: The reference count is stored in the object itself (in `vsg::Object`) - **Thread-safe**: Uses atomic operations for the reference count - **Efficient**: No separate control block allocation - **Automatic memory management**: Objects are deleted when reference count reaches zero

Object Creation Pattern

```
// Always use the static create() method
auto node = vsg::Node::create();

// Constructor parameters can be passed to create()
auto transform = vsg::MatrixTransform::create(matrix);
```

Reference Counting Behavior

```
{
    vsg::ref_ptr<vsg::Group> group = vsg::Group::create();  // ref count = 1
    {
        vsg::ref_ptr<vsg::Group> another = group;           // ref count = 2
    }                                                        // ref count = 1
}                                                            // ref count = 0, object deleted
```

Visitor Pattern

The visitor pattern is fundamental to VSG's design, enabling type-safe traversal and processing of scene graphs.

Basic Visitor Usage

```
// Create a visitor to count node types
struct NodeCounter : public vsg::ConstVisitor
{
    std::map<const char*, uint32_t> counts;

    void apply(const vsg::Object& object) override
    {
        ++counts[object.className()];
        object.traverse(*this);  // Continue traversal
    }
};

// Use the visitor
NodeCounter counter;
scene->accept(counter);

// Or use the convenience template
auto counts = vsg::visit<NodeCounter>(scene).counts;
```

Custom Visitors

```
// Visitor for modifying transforms
class TransformVisitor : public vsg::Visitor
{
public:
    void apply(vsg::MatrixTransform& transform) override
    {
        transform.matrix = vsg::rotate(angle, axis) * transform.matrix;
        transform.traverse(*this);
    }

    void apply(vsg::Group& group) override
    {
        group.traverse(*this);  // Just traverse children
    }
};
```

Visitor Types

- **vsg::Visitor** - Can modify the scene graph
- **vsg::ConstVisitor** - Read-only traversal
- **vsg::ArrayState** - Specialized for array dispatching
- **vsg::Intersector** - For intersection testing

Type System

VSG provides a robust type system with RTTI support and safe casting.

Object Hierarchy

```
vsg::Object
├── vsg::Data
│   ├── vsg::Array
│   └── vsg::Value<T>
├── vsg::Node
│   ├── vsg::Group
│   ├── vsg::Transform
│   └── vsg::Geometry
└── vsg::StateComponent
    ├── vsg::StateCommand
    └── vsg::StateAttribute
```

Type Identification

```cpp
// Get class name
const char* className = object->className();

// Type-safe casting
if (auto group = node->cast<vsg::Group>()) {
    // node is a Group
    group->addChild(child);
}

// Check type compatibility
if (node->is_compatible(typeid(vsg::Transform))) {
    // node is or derives from Transform
}
```

Creating Custom Types

```cpp
// Use vsg::Inherit for proper RTTI and ref counting
class MyNode : public vsg::Inherit<vsg::Group, MyNode>
{
public:
    float customProperty = 0.0f;

protected:
    ~MyNode() = default;  // Protected destructor
};

// Custom visitor support
class MyVisitor : public vsg::Visitor
{
public:
    void apply(vsg::Group& group) override
    {
        if (auto myNode = group.cast<MyNode>()) {
            apply(*myNode);
        } else {
            group.traverse(*this);
        }
    }

    virtual void apply(MyNode& node)
    {
        // Process custom node
        node.customProperty *= 2.0f;
        node.traverse(*this);
    }
};
```

Value Types

```cpp
// Attach typed values to objects
object->setValue("speed", 100.0f);
object->setValue("name", std::string("Player"));

// Retrieve values
float speed = 0.0f;
if (object->getValue("speed", speed)) {
    // Use speed value
}

// Custom value types
struct GameState {
    int level;
    float health;
};
using GameStateValue = vsg::Value<GameState>;
EVSG_type_name(GameStateValue);  // Register for serialization
```

Memory Management

Allocators

VSG provides a sophisticated memory allocation system designed for high-performance scene graph applications. The allocator system uses block-based allocation with different pools for different types of objects.

Allocator Types

```
// Get the global allocator instance
auto& allocator = vsg::Allocator::instance();

// Use intrusive allocator with custom alignment
vsg::Allocator::instance().reset(
    new vsg::IntrusiveAllocator(std::move(vsg::Allocator::instance()), 16));

// Use standard allocator (fallback to new/delete)
vsg::Allocator::instance().reset(
    new StdAllocator(std::move(vsg::Allocator::instance())));
```

Allocation Affinity

VSG uses different memory pools for different object types to optimize cache performance:

```
// Configure block sizes for different allocation types
allocator->setBlockSize(vsg::ALLOCATOR_AFFINITY_OBJECTS, 65536);  // General objects
allocator->setBlockSize(vsg::ALLOCATOR_AFFINITY_NODES, 131072);   // Scene nodes
allocator->setBlockSize(vsg::ALLOCATOR_AFFINITY_DATA, 1048576);   // Large data arrays
```

Memory Management

```
// Report memory usage
vsg::Allocator::instance()->report(std::cout);

// Clean up empty memory blocks (optional)
size_t freed = vsg::Allocator::instance()->deleteEmptyMemoryBlocks();

// Query memory statistics
size_t available = allocator->totalAvailableSize();
size_t reserved = allocator->totalReservedSize();
size_t total = allocator->totalMemorySize();
```

Key benefits: - **Block-based allocation**: Reduces fragmentation and allocation overhead - **Memory pools**: Different pools for different object types improve cache locality - **Thread-safe**: All allocations are thread-safe - **Efficient reuse**: Memory blocks are reused, minimizing system allocations - **Configurable**: Block sizes and alignment can be tuned for specific workloads

Arrays and Data Types

VSG provides type-safe array classes and a comprehensive type system that maps directly to Vulkan formats.

Array Types

```
// Create typed arrays
auto floats = vsg::floatArray::create(10);
auto colors = vsg::vec4Array::create(100);
auto vertices = vsg::vec3Array::create(1000);
auto texCoords = vsg::vec2Array::create({{0.0f, 0.0f}, {1.0f, 0.0f}, {1.0f, 1.0f}});

// Arrays support STL algorithms
std::for_each(floats->begin(), floats->end(), [](float& v) { v = 1.0f; });

// Index-based access
for (size_t i = 0; i < vertices->size(); ++i) {
    (*vertices)[i] = vsg::vec3(x, y, z);
}

// Range-based for loops
for (auto& vertex : *vertices) {
    vertex.z = 0.0f;
}
```

Vector and Matrix Types

```
// Float precision vectors
vsg::vec2   // 2D vector (float)
vsg::vec3   // 3D vector (float)
vsg::vec4   // 4D vector (float)

// Double precision vectors
vsg::dvec2  // 2D vector (double)
vsg::dvec3  // 3D vector (double)
vsg::dvec4  // 4D vector (double)

// Integer vectors
vsg::ivec2, vsg::ivec3, vsg::ivec4    // signed int
vsg::uivec2, vsg::uivec3, vsg::uivec4 // unsigned int

// Byte vectors (for colors)
vsg::ubvec4 // unsigned byte vec4 (common for RGBA colors)

// Matrices
vsg::mat4   // 4x4 matrix (float)
vsg::dmat4  // 4x4 matrix (double)
```

Vulkan Format Mapping

VSG types map directly to Vulkan formats for efficient GPU transfer:

```
vsg::vec3   -> VK_FORMAT_R32G32B32_SFLOAT
vsg::vec4   -> VK_FORMAT_R32G32B32A32_SFLOAT
vsg::ubvec4 -> VK_FORMAT_R8G8B8A8_UNORM
float       -> VK_FORMAT_R32_SFLOAT
uint32_t    -> VK_FORMAT_R32_UINT
```

Mathematical Operations

```
// Angle conversion
float radians = vsg::radians(degrees);
float degrees = vsg::degrees(radians);

// Vector operations
float len = vsg::length(vec);
vsg::vec3 normalized = vsg::normalize(vec);
float dotProduct = vsg::dot(vec1, vec2);
vsg::vec3 crossProduct = vsg::cross(vec1, vec2);

// Matrix transformations
auto rotMatrix = vsg::rotate(angle, axis);
auto transMatrix = vsg::translate(offset);
auto scaleMatrix = vsg::scale(factors);
```

Scene Graph

The VSG scene graph provides a hierarchical structure for organizing 3D content, with efficient node management, state inheritance, and flexible rendering control.

Node Hierarchy and Types

VSG's scene graph is built on a comprehensive node hierarchy:

```
vsg::Object                     // Base class for all VSG objects
├── vsg::Node                   // Base class for scene graph nodes
│   ├── vsg::Group              // Container for multiple children
│   │   ├── vsg::QuadGroup      // Spatial organization with quadrants
│   │   ├── vsg::LOD            // Level-of-detail management
│   │   └── vsg::Switch         // Conditional rendering control
│   ├── vsg::Transform          // Apply transformations to children
│   │   └── vsg::MatrixTransform // Matrix-based transformations
│   ├── vsg::StateGroup         // Apply state changes to children
│   └── vsg::Commands           // Renderable leaf nodes
```

Node Management Operations

Adding and Removing Nodes

```
// Create root group
auto root = vsg::Group::create();

// Add children
root->addChild(childNode);
root->addChild(transform);

// Remove specific child
root->removeChild(childNode);

// Clear all children
root->children.clear();

// Batch operations for performance
root->children.reserve(expectedSize);
for (auto& node : nodeList) {
    root->addChild(node);
}

// Check if node is a child
if (std::find(root->children.begin(), root->children.end(), node) != root->children.end()) {
    // Node is a child
}
```

Dynamic Scene Modification

```
// Runtime scene modification
class SceneManager
{
public:
    void addObject(vsg::ref_ptr<vsg::Node> object, const vsg::dvec3& position)
    {
        auto transform = vsg::MatrixTransform::create();
        transform->matrix = vsg::translate(position);
        transform->addChild(object);

        dynamicGroup->addChild(transform);
    }

    void removeObject(vsg::ref_ptr<vsg::Node> object)
    {
        // Find and remove object from scene
        auto it = std::find_if(dynamicGroup->children.begin(), dynamicGroup->children.end(),
            [object](vsg::ref_ptr<vsg::Node> child) {
                if (auto transform = child.cast<vsg::MatrixTransform>()) {
                    return !transform->children.empty() && transform->children[0] == object;
                }
                return false;
            });

        if (it != dynamicGroup->children.end()) {
            dynamicGroup->removeChild(*it);
        }
    }

private:
    vsg::ref_ptr<vsg::Group> dynamicGroup = vsg::Group::create();
};
```

Groups and Transforms

Transform Nodes

```
// Matrix-based transformations
```

```cpp
auto transform = vsg::MatrixTransform::create();

// Static transformation
transform->matrix = vsg::translate(vsg::vec3(10.0f, 0.0f, 0.0f)) *
                   vsg::rotate(vsg::radians(45.0f), vsg::vec3(0.0f, 0.0f, 1.0f)) *
                   vsg::scale(vsg::vec3(2.0f, 2.0f, 2.0f));

// Dynamic animation
while (viewer->advanceToNextFrame()) {
    float time = std::chrono::duration<float>(viewer->getFrameStamp()->time - viewer->start_point()).count();

    // Rotation animation
    transform->matrix = vsg::rotate(time * vsg::radians(90.0f), vsg::vec3(0.0f, 0.0f, 1.0f));

    // Complex animation combining transformations
    auto rotation = vsg::rotate(time * vsg::radians(45.0f), vsg::vec3(0.0f, 1.0f, 0.0f));
    auto translation = vsg::translate(vsg::vec3(sin(time) * 5.0f, 0.0f, 0.0f));
    transform->matrix = translation * rotation;

    viewer->update();
    viewer->recordAndSubmit();
    viewer->present();
}
```

Specialized Groups

```cpp
// QuadGroup for spatial organization
auto quadGroup = vsg::QuadGroup::create();
quadGroup->addChild(0, northWestChild);   // NW quadrant
quadGroup->addChild(1, northEastChild);   // NE quadrant
quadGroup->addChild(2, southEastChild);   // SE quadrant
quadGroup->addChild(3, southWestChild);   // SW quadrant

// Switch nodes for conditional rendering
auto switchNode = vsg::Switch::create();
switchNode->addChild(false, winterScene);  // Initially disabled
switchNode->addChild(true, summerScene);   // Initially enabled

// Toggle visibility
bool showWinter = true;
switchNode->setChild(0, showWinter, winterScene);
switchNode->setChild(1, !showWinter, summerScene);

// Enable/disable all children
switchNode->setAllChildren(false);  // Hide all
switchNode->setAllChildren(true);   // Show all
```

Level of Detail (LOD)

```cpp
// LOD for performance optimization
auto lodNode = vsg::LOD::create();

// Add LOD levels with distance ranges
lodNode->addChild(vsg::LODRange{0.0, 100.0}, highDetailModel);     // Close range
lodNode->addChild(vsg::LODRange{100.0, 500.0}, mediumDetailModel); // Medium range
lodNode->addChild(vsg::LODRange{500.0, DBL_MAX}, lowDetailModel);  // Far range

// Set LOD center point for distance calculations
lodNode->center = vsg::dvec3(0.0, 0.0, 0.0);

// Example with Builder-generated LOD models
vsg::Builder builder;
vsg::GeometryInfo geomInfo;
vsg::StateInfo stateInfo;

// High detail sphere (many vertices)
geomInfo.dx = geomInfo.dy = geomInfo.dz = vsg::vec3(2.0f, 2.0f, 2.0f);
auto highDetailSphere = builder.createSphere(geomInfo, stateInfo);

// Medium detail sphere
```

```
geomInfo.dx = geomInfo.dy = geomInfo.dz = vsg::vec3(1.0f, 1.0f, 1.0f);
auto mediumDetailSphere = builder.createSphere(geomInfo, stateInfo);

// Low detail sphere (fewer vertices)
geomInfo.dx = geomInfo.dy = geomInfo.dz = vsg::vec3(0.5f, 0.5f, 0.5f);
auto lowDetailSphere = builder.createSphere(geomInfo, stateInfo);

lodNode->addChild(vsg::LODRange{0.0, 50.0}, highDetailSphere);
lodNode->addChild(vsg::LODRange{50.0, 200.0}, mediumDetailSphere);
lodNode->addChild(vsg::LODRange{200.0, DBL_MAX}, lowDetailSphere);
```

Creating New Primitives and Geometry

Using VSG Builder for Standard Primitives

```
// Initialize builder and configuration
vsg::Builder builder;
vsg::GeometryInfo geomInfo;
vsg::StateInfo stateInfo;

// Configure geometry properties
geomInfo.color = vsg::vec4{1.0f, 0.0f, 0.0f, 1.0f};  // Red color
geomInfo.position = vsg::vec3{0.0f, 0.0f, 0.0f};     // Origin
geomInfo.dx = vsg::vec3{2.0f, 0.0f, 0.0f};           // X-axis dimension
geomInfo.dy = vsg::vec3{0.0f, 2.0f, 0.0f};           // Y-axis dimension
geomInfo.dz = vsg::vec3{0.0f, 0.0f, 2.0f};           // Z-axis dimension

// Create standard primitives
auto sphere = builder.createSphere(geomInfo, stateInfo);
auto cylinder = builder.createCylinder(geomInfo, stateInfo);
auto cone = builder.createCone(geomInfo, stateInfo);
auto box = builder.createBox(geomInfo, stateInfo);
auto capsule = builder.createCapsule(geomInfo, stateInfo);

// Apply transformations during creation
geomInfo.transform = vsg::translate(vsg::vec3(5.0f, 0.0f, 0.0f)) * vsg::scale(0.5f, 0.5f, 0.5f);
auto scaledSphere = builder.createSphere(geomInfo, stateInfo);
```

Manual Geometry Creation

```
// Custom vertex data - separate arrays
auto vertices = vsg::vec3Array::create({
    {-1.0f, -1.0f, 0.0f},  // Bottom-left
    { 1.0f, -1.0f, 0.0f},  // Bottom-right
    { 1.0f,  1.0f, 0.0f},  // Top-right
    {-1.0f,  1.0f, 0.0f},  // Top-left
    { 0.0f,  0.0f, 1.0f}   // Peak (for pyramid)
});

auto colors = vsg::vec3Array::create({
    {1.0f, 0.0f, 0.0f},    // Red
    {0.0f, 1.0f, 0.0f},    // Green
    {0.0f, 0.0f, 1.0f},    // Blue
    {1.0f, 1.0f, 0.0f},    // Yellow
    {1.0f, 0.0f, 1.0f}     // Magenta
});

auto texCoords = vsg::vec2Array::create({
    {0.0f, 0.0f}, {1.0f, 0.0f}, {1.0f, 1.0f}, {0.0f, 1.0f}, {0.5f, 0.5f}
});

// Create indices for different primitive types
auto quadIndices = vsg::ushortArray::create({
    0, 1, 2, 2, 3, 0  // Two triangles forming a quad
});

auto pyramidIndices = vsg::ushortArray::create({
    0, 1, 4,  // Base triangle 1
    1, 2, 4,  // Base triangle 2
    2, 3, 4,  // Base triangle 3
```

```
    3, 0, 4   // Base triangle 4
});
```

Interleaved Vertex Data (Performance Optimized)

```
// Interleaved vertex data for better cache locality
auto attributeArray = vsg::floatArray::create({
    // x,    y,    z,     r,    g,    b,     s,    t
    -0.5f, -0.5f, 0.0f,  1.0f, 0.0f, 0.0f,  0.0f, 0.0f,  // Vertex 0
     0.5f, -0.5f, 0.0f,  0.0f, 1.0f, 0.0f,  1.0f, 0.0f,  // Vertex 1
     0.5f,  0.5f, 0.0f,  0.0f, 0.0f, 1.0f,  1.0f, 1.0f,  // Vertex 2
    -0.5f,  0.5f, 0.0f,  1.0f, 1.0f, 1.0f,  0.0f, 1.0f   // Vertex 3
});

// Vertex input configuration for interleaved data
vsg::VertexInputState::Bindings vertexBindings{
    VkVertexInputBindingDescription{0, 32, VK_VERTEX_INPUT_RATE_VERTEX}  // 32 bytes stride
};

vsg::VertexInputState::Attributes vertexAttributes{
    VkVertexInputAttributeDescription{0, 0, VK_FORMAT_R32G32B32_SFLOAT, 0},   // position: offset 0
    VkVertexInputAttributeDescription{1, 0, VK_FORMAT_R32G32B32_SFLOAT, 12},  // color: offset 12
    VkVertexInputAttributeDescription{2, 0, VK_FORMAT_R32G32_SFLOAT, 24}      // texcoord: offset 24
};

// Create draw commands
auto drawCommands = vsg::Commands::create();
drawCommands->addChild(vsg::BindVertexBuffers::create(0, vsg::DataList{attributeArray}));
drawCommands->addChild(vsg::BindIndexBuffer::create(indices));
drawCommands->addChild(vsg::DrawIndexed::create(indexCount, 1, 0, 0, 0));
```

Creating Custom Shaders for Primitives

Basic Shader Creation

```
// Load shaders from SPIR-V files
auto vertexShader = vsg::ShaderStage::read(VK_SHADER_STAGE_VERTEX_BIT, "main",
    vsg::findFile("shaders/vert_PushConstants.spv", searchPaths));
auto fragmentShader = vsg::ShaderStage::read(VK_SHADER_STAGE_FRAGMENT_BIT, "main",
    vsg::findFile("shaders/frag_PushConstants.spv", searchPaths));

// Create shaders from source code
auto vertexShader = vsg::ShaderStage::create(VK_SHADER_STAGE_VERTEX_BIT, "main", vertexSource);
auto fragmentShader = vsg::ShaderStage::create(VK_SHADER_STAGE_FRAGMENT_BIT, "main", fragmentSource);
```

Skybox Shader Example

```
// Skybox vertex shader - removes translation for infinite distance effect
const auto skybox_vert = R"(
#version 450

layout(push_constant) uniform PushConstants {
    mat4 projection;
    mat4 modelView;
} pc;

layout(location = 0) in vec3 vsg_Vertex;
layout(location = 0) out vec3 UVW;

void main() {
    UVW = vsg_Vertex;  // Use vertex position as texture coordinate

    // Remove translation component to keep skybox centered on camera
    mat4 modelView = pc.modelView;
    modelView[3] = vec4(0.0, 0.0, 0.0, 1.0);  // Zero out translation

    vec4 pos = pc.projection * modelView * vec4(vsg_Vertex, 1.0);
    gl_Position = vec4(pos.xy, 0.0, pos.w);  // Set z to far plane
}
```

```
)";

// Skybox fragment shader - samples cube map texture
const auto skybox_frag = R"(
#version 450

layout(binding = 0) uniform samplerCube envMap;

layout(location = 0) in vec3 UVW;
layout(location = 0) out vec4 outColor;

void main() {
    outColor = textureLod(envMap, UVW, 0);  // Sample cube map
}
)";

// Create skybox shaders
auto skyboxVertShader = vsg::ShaderStage::create(VK_SHADER_STAGE_VERTEX_BIT, "main", skybox_vert);
auto skyboxFragShader = vsg::ShaderStage::create(VK_SHADER_STAGE_FRAGMENT_BIT, "main", skybox_frag);
```

Complete Graphics Pipeline Creation

```
// Graphics pipeline setup with custom shaders
vsg::GraphicsPipelineStates pipelineStates{
    vsg::VertexInputState::create(vertexBindings, vertexAttributes),
    vsg::InputAssemblyState::create(),        // Primitive assembly
    vsg::RasterizationState::create(),        // Rasterization settings
    vsg::MultisampleState::create(),          // MSAA configuration
    vsg::ColorBlendState::create(),           // Color blending
    vsg::DepthStencilState::create()          // Depth testing
};

// Custom rasterization state for skybox
auto rasterState = vsg::RasterizationState::create();
rasterState->cullMode = VK_CULL_MODE_FRONT_BIT;  // Cull front faces for skybox

// Custom depth state for skybox
auto depthState = vsg::DepthStencilState::create();
depthState->depthTestEnable = VK_TRUE;
depthState->depthWriteEnable = VK_FALSE;
depthState->depthCompareOp = VK_COMPARE_OP_GREATER_OR_EQUAL;  // Reverse depth

// Push constants for efficient matrix transfer
vsg::PushConstantRanges pushConstantRanges{
    {VK_SHADER_STAGE_VERTEX_BIT, 0, 128}  // projection, view, and model matrices
};

auto pipelineLayout = vsg::PipelineLayout::create(
    vsg::DescriptorSetLayouts{descriptorSetLayout},
    pushConstantRanges);

auto graphicsPipeline = vsg::GraphicsPipeline::create(pipelineLayout,
    vsg::ShaderStages{vertexShader, fragmentShader}, pipelineStates);
```

Switching Between 2D Tiled Maps and 3D Tiled Globes

Projection Type Management

```
// Camera setup for different projection modes
class ProjectionManager
{
public:
    void setupCameras(vsg::ref_ptr<vsg::Node> scene)
    {
        // Compute scene bounds for camera positioning
        vsg::ComputeBounds computeBounds;
        scene->accept(computeBounds);
        vsg::dvec3 centre = (computeBounds.bounds.min + computeBounds.bounds.max) * 0.5;
        double radius = vsg::length(computeBounds.bounds.max - computeBounds.bounds.min) * 0.6;
```

```
        // Create view matrix
        auto lookAt = vsg::LookAt::create(
            centre + vsg::dvec3(0.0, -radius * 3.5, 0.0),
            centre,
            vsg::dvec3(0.0, 0.0, 1.0));

        // Check for geospatial data (ellipsoid model)
        auto ellipsoidModel = scene->getRefObject<vsg::EllipsoidModel>("EllipsoidModel");

        if (ellipsoidModel) {
            // 3D Globe projection for geospatial data
            perspective3D = vsg::EllipsoidPerspective::create(
                lookAt,
                ellipsoidModel,
                30.0,                    // Field of view
                aspectRatio,
                0.001,                   // Near/far ratio
                0.0                      // Horizon mountain height
            );

            // 2D Map projection (orthographic)
            perspective2D = vsg::Orthographic::create(
                -radius, radius,         // Left, right
                -radius, radius,         // Bottom, top
                0.001 * radius,          // Near
                radius * 10.0            // Far
            );
        } else {
            // Standard 3D perspective
            perspective3D = vsg::Perspective::create(
                30.0,                    // Field of view
                aspectRatio,
                0.001 * radius,          // Near
                radius * 4.5             // Far
            );

            // 2D orthographic view
            perspective2D = vsg::Orthographic::create(
                -radius, radius, -radius, radius,
                0.001 * radius, radius * 4.5);
        }

        // Create cameras
        camera3D = vsg::Camera::create(perspective3D, lookAt, viewport);
        camera2D = vsg::Camera::create(perspective2D, lookAt, viewport);
    }

    void switch2D3D(bool use3D)
    {
        is3DMode = use3D;
        activeCamera = use3D ? camera3D : camera2D;

        // Update render graph with new camera
        if (renderGraph) {
            renderGraph->camera = activeCamera;
        }
    }

    bool is3D() const { return is3DMode; }
    vsg::ref_ptr<vsg::Camera> getActiveCamera() const { return activeCamera; }

private:
    vsg::ref_ptr<vsg::ProjectionMatrix> perspective3D;
    vsg::ref_ptr<vsg::ProjectionMatrix> perspective2D;
    vsg::ref_ptr<vsg::Camera> camera3D;
    vsg::ref_ptr<vsg::Camera> camera2D;
    vsg::ref_ptr<vsg::Camera> activeCamera;
    vsg::ref_ptr<vsg::RenderGraph> renderGraph;
    double aspectRatio = 1.0;
    bool is3DMode = true;
};
```

Scene Organization for Different Views

```
// Organize scene for 2D/3D switching
class TiledSceneManager
{
public:
    void setupScene()
    {
        root = vsg::Group::create();

        // Switch node for different projection modes
        projectionSwitch = vsg::Switch::create();

        // 2D tiled map view
        auto map2DView = vsg::StateGroup::create();
        map2DView->add(create2DTileShader());
        map2DView->addChild(tileGeometry2D);
        projectionSwitch->addChild(false, map2DView);  // Initially disabled

        // 3D globe view
        auto globe3DView = vsg::StateGroup::create();
        globe3DView->add(create3DGlobeShader());
        globe3DView->addChild(tileGeometry3D);
        projectionSwitch->addChild(true, globe3DView);   // Initially enabled

        root->addChild(projectionSwitch);

        // Common elements (UI, overlays)
        auto commonElements = vsg::Group::create();
        commonElements->addChild(createUI());
        commonElements->addChild(createOverlays());
        root->addChild(commonElements);
    }

    void toggleProjection()
    {
        bool current2D = projectionSwitch->children[0].mask != 0;

        // Toggle visibility
        projectionSwitch->children[0].mask = current2D ? 0 : ~0;  // 2D view
        projectionSwitch->children[1].mask = current2D ? ~0 : 0;  // 3D view

        // Update camera projection
        projectionManager.switch2D3D(!current2D);
    }

    void updateTileLevel(int zoomLevel)
    {
        // Update both 2D and 3D tile representations
        update2DTiles(zoomLevel);
        update3DTiles(zoomLevel);
    }

private:
    vsg::ref_ptr<vsg::Group> root;
    vsg::ref_ptr<vsg::Switch> projectionSwitch;
    vsg::ref_ptr<vsg::Node> tileGeometry2D;
    vsg::ref_ptr<vsg::Node> tileGeometry3D;
    ProjectionManager projectionManager;

    vsg::ref_ptr<vsg::StateGroup> create2DTileShader()
    {
        // Shader setup for 2D tile rendering
        auto stateGroup = vsg::StateGroup::create();
        // Add 2D-specific shaders and state
        return stateGroup;
    }

    vsg::ref_ptr<vsg::StateGroup> create3DGlobeShader()
    {
```

```
        // Shader setup for 3D globe rendering
        auto stateGroup = vsg::StateGroup::create();
        // Add 3D globe-specific shaders and state
        return stateGroup;
    }
};
```

State Management and Inheritance

StateGroup Hierarchy

```
// Hierarchical state management
auto rootState = vsg::StateGroup::create();

// Material state
auto materialState = vsg::StateGroup::create();
materialState->add(vsg::BindGraphicsPipeline::create(materialPipeline));

// Texture state (inherits material)
auto textureState = vsg::StateGroup::create();
textureState->add(vsg::BindDescriptorSet::create(
    VK_PIPELINE_BIND_POINT_GRAPHICS,
    pipelineLayout,
    0,
    textureDescriptorSet));

// Geometry (inherits both material and texture)
textureState->addChild(geometry);
materialState->addChild(textureState);
rootState->addChild(materialState);

// State overrides
auto highlightState = vsg::StateGroup::create();
highlightState->add(vsg::BindGraphicsPipeline::create(highlightPipeline));
highlightState->addChild(selectedGeometry);  // Overrides material pipeline
rootState->addChild(highlightState);
```

Dynamic State Updates

```
// Runtime state modification
class DynamicStateManager
{
public:
    void updateMaterial(vsg::ref_ptr<vsg::StateGroup> stateGroup, const Material& material)
    {
        // Update shader uniforms
        auto uniform = vsg::vec4Value::create(material.diffuseColor);
        stateGroup->setValue("diffuseColor", uniform);

        // Update textures
        if (material.diffuseTexture) {
            auto texture = vsg::DescriptorImage::create(
                vsg::ImageInfo::create(sampler, material.diffuseTexture));
            // Update descriptor set
        }
    }

    void setLighting(bool enabled)
    {
        if (enabled) {
            lightingState->add(vsg::BindGraphicsPipeline::create(litPipeline));
        } else {
            lightingState->add(vsg::BindGraphicsPipeline::create(unlitPipeline));
        }
    }

    void setWireframe(bool wireframe)
    {
        auto rasterState = vsg::RasterizationState::create();
        rasterState->polygonMode = wireframe ? VK_POLYGON_MODE_LINE : VK_POLYGON_MODE_FILL;
```

```
        wireframeState->stateCommands.clear();
        wireframeState->add(rasterState);
    }

private:
    vsg::ref_ptr<vsg::StateGroup> lightingState;
    vsg::ref_ptr<vsg::StateGroup> wireframeState;
    vsg::ref_ptr<vsg::GraphicsPipeline> litPipeline;
    vsg::ref_ptr<vsg::GraphicsPipeline> unlitPipeline;
};
```

Rendering

Graphics Pipeline and Command Buffers

VSG provides both high-level convenience and low-level control over Vulkan graphics pipelines and command recording.

Manual Graphics Pipeline Creation

For applications requiring fine-grained control, VSG allows direct graphics pipeline creation:

```
// Vertex input state with multiple attribute streams
vsg::VertexInputState::Bindings vertexBindingsDescriptions{
    VkVertexInputBindingDescription{0, sizeof(vsg::vec3), VK_VERTEX_INPUT_RATE_VERTEX}, // vertices
    VkVertexInputBindingDescription{1, sizeof(vsg::vec3), VK_VERTEX_INPUT_RATE_VERTEX}, // colors
    VkVertexInputBindingDescription{2, sizeof(vsg::vec2), VK_VERTEX_INPUT_RATE_VERTEX}  // texcoords
};

vsg::VertexInputState::Attributes vertexAttributeDescriptions{
    VkVertexInputAttributeDescription{0, 0, VK_FORMAT_R32G32B32_SFLOAT, 0},
    VkVertexInputAttributeDescription{1, 1, VK_FORMAT_R32G32B32_SFLOAT, 0},
    VkVertexInputAttributeDescription{2, 2, VK_FORMAT_R32G32_SFLOAT, 0}
};

// Complete pipeline state creation
vsg::GraphicsPipelineStates pipelineStates{
    vsg::VertexInputState::create(vertexBindingsDescriptions, vertexAttributeDescriptions),
    vsg::InputAssemblyState::create(),    // Primitive assembly
    vsg::RasterizationState::create(),    // Rasterization settings
    vsg::MultisampleState::create(),      // MSAA configuration
    vsg::ColorBlendState::create(),       // Color blending
    vsg::DepthStencilState::create()      // Depth testing
};

auto graphicsPipeline = vsg::GraphicsPipeline::create(pipelineLayout,
    vsg::ShaderStages{vertexShader, fragmentShader}, pipelineStates);
```

Direct Draw Commands

VSG allows manual command buffer construction for performance-critical applications:

```
// Create explicit draw commands
auto drawCommands = vsg::Commands::create();
drawCommands->addChild(vsg::BindVertexBuffers::create(0, vsg::DataList{vertices, colors, texcoords}));
drawCommands->addChild(vsg::BindIndexBuffer::create(indices));
drawCommands->addChild(vsg::DrawIndexed::create(indexCount, instanceCount, firstIndex, vertexOffset, firstInstance));

// Combine with state management
auto stateGroup = vsg::StateGroup::create();
stateGroup->add(vsg::BindGraphicsPipeline::create(graphicsPipeline));
stateGroup->add(vsg::BindDescriptorSet::create(VK_PIPELINE_BIND_POINT_GRAPHICS, pipelineLayout, 0, descriptorSet));
stateGroup->addChild(drawCommands);
```

Push Constants for Matrices

Efficient matrix data transfer using Vulkan push constants:

```
vsg::PushConstantRanges pushConstantRanges{
    {VK_SHADER_STAGE_VERTEX_BIT, 0, 128} // projection, view, and model matrices
```

```
};

auto pipelineLayout = vsg::PipelineLayout::create(
    vsg::DescriptorSetLayouts{descriptorSetLayout},
    pushConstantRanges);
```

GPU Queries and Performance Analysis

VSG provides comprehensive support for Vulkan query objects, enabling GPU-based performance analysis and visibility testing.

Occlusion Queries

Determine visibility of rendered geometry for performance optimization:

```
// Create occlusion query pool
auto query_pool = vsg::QueryPool::create();
query_pool->queryType = VK_QUERY_TYPE_OCCLUSION;
query_pool->queryCount = 1;

// Embed in command graph
commandGraph->addChild(vsg::ResetQueryPool::create(query_pool));
commandGraph->addChild(vsg::BeginQuery::create(query_pool, 0, 0));
commandGraph->addChild(renderGraph);  // Render scene
commandGraph->addChild(vsg::EndQuery::create(query_pool, 0));

// Retrieve results
std::vector<uint64_t> results(1);
if (query_pool->getResults(results) == VK_SUCCESS)
{
    uint64_t visibleFragments = results[0];
    // Use for LOD selection or culling decisions
}
```

Timestamp Queries

Measure precise GPU execution timing for performance profiling:

```
// Create timestamp query pool
auto timestampPool = vsg::QueryPool::create();
timestampPool->queryType = VK_QUERY_TYPE_TIMESTAMP;
timestampPool->queryCount = 2;

// Check device timestamp support
const auto& limits = physicalDevice->getProperties().limits;
if (!limits.timestampComputeAndGraphics)
{
    // Handle unsupported devices
    return;
}

// Calculate time scaling factor
double timestampScale = 1e-6 * static_cast<double>(limits.timestampPeriod); // Convert to milliseconds

// Embed timing in command graph
commandGraph->addChild(vsg::ResetQueryPool::create(timestampPool));
commandGraph->addChild(vsg::WriteTimestamp::create(VK_PIPELINE_STAGE_TOP_OF_PIPE_BIT, timestampPool, 0));
commandGraph->addChild(renderGraph);  // Timed operations
commandGraph->addChild(vsg::WriteTimestamp::create(VK_PIPELINE_STAGE_BOTTOM_OF_PIPE_BIT, timestampPool, 1));

// Calculate elapsed time
std::vector<uint64_t> timestamps(2);
if (timestampPool->getResults(timestamps) == VK_SUCCESS)
{
    double gpuTime = timestampScale * static_cast<double>(timestamps[1] - timestamps[0]);
    std::cout << "GPU render time: " << gpuTime << "ms" << std::endl;
}
```

Vertex Data Organization

VSG supports both separate and interleaved vertex arrays for optimal memory access:

```
// Interleaved vertex data for better cache locality
auto interleavedData = vsg::floatArray::create({
    // x,    y,    z,     r,    g,    b,     s,    t
    -0.5f, -0.5f, 0.0f,  1.0f, 0.0f, 0.0f,  0.0f, 0.0f,  // Vertex 0
     0.5f, -0.5f, 0.0f,  0.0f, 1.0f, 0.0f,  1.0f, 0.0f,  // Vertex 1
    // ... more vertices
});

// Single binding with multiple attributes
vsg::VertexInputState::Bindings vertexBindings{
    VkVertexInputBindingDescription{0, 32, VK_VERTEX_INPUT_RATE_VERTEX} // 32 bytes stride
};

vsg::VertexInputState::Attributes vertexAttributes{
    VkVertexInputAttributeDescription{0, 0, VK_FORMAT_R32G32B32_SFLOAT, 0},  // position: offset 0
    VkVertexInputAttributeDescription{1, 0, VK_FORMAT_R32G32B32_SFLOAT, 12}, // color: offset 12
    VkVertexInputAttributeDescription{2, 0, VK_FORMAT_R32G32_SFLOAT, 24},    // texcoord: offset 24
};

// Single vertex buffer binding
drawCommands->addChild(vsg::BindVertexBuffers::create(0, vsg::DataList{interleavedData}));
```

Basic Setup

Creating a VSG application involves several key components working together.

Minimal Application Structure

```
#include <vsg/all.h>
#include <vsgXchange/all.h>

int main(int argc, char** argv)
{
    // Parse command line
    vsg::CommandLine arguments(&argc, argv);

    // Set up file loading options
    auto options = vsg::Options::create(vsgXchange::all::create());

    // Load scene
    auto scene = vsg::read_cast<vsg::Node>("model.vsgt", options);
    if (!scene) return 1;

    // Create viewer and window
    auto viewer = vsg::Viewer::create();
    auto window = vsg::Window::create(vsg::WindowTraits::create());
    viewer->addWindow(window);

    // Set up camera
    auto camera = createCameraForScene(scene);

    // Create rendering command graph
    auto commandGraph = vsg::createCommandGraphForView(window, camera, scene);
    viewer->assignRecordAndSubmitTaskAndPresentation({commandGraph});

    // Compile and transfer to GPU
    viewer->compile();

    // Main loop
    while (viewer->advanceToNextFrame())
    {
        viewer->handleEvents();
        viewer->update();
        viewer->recordAndSubmit();
        viewer->present();
    }

    return 0;
}
```

Window Configuration

```
auto traits = vsg::WindowTraits::create();
traits->windowTitle = "My VSG Application";
traits->width = 1920;
traits->height = 1080;
traits->fullscreen = false;
traits->samples = VK_SAMPLE_COUNT_8_BIT;  // MSAA
traits->debugLayer = true;  // Enable validation layers
traits->apiDumpLayer = false;

auto window = vsg::Window::create(traits);
```

Camera Setup with Scene Bounds

```
// Calculate scene bounds
vsg::ComputeBounds computeBounds;
scene->accept(computeBounds);
auto bounds = computeBounds.bounds;
vsg::dvec3 centre = (bounds.min + bounds.max) * 0.5;
double radius = vsg::length(bounds.max - bounds.min) * 0.6;

// Create view matrix (LookAt)
auto lookAt = vsg::LookAt::create(
    centre + vsg::dvec3(0.0, -radius * 3.5, 0.0),  // eye position
    centre,                                         // look at point
    vsg::dvec3(0.0, 0.0, 1.0)                      // up vector
);

// Create projection matrix
auto perspective = vsg::Perspective::create(
    30.0,                                          // vertical FOV
    static_cast<double>(window->extent2D().width) /
    static_cast<double>(window->extent2D().height), // aspect ratio
    radius * 0.001,                                // near plane
    radius * 4.5                                   // far plane
);

// Combine into camera
auto camera = vsg::Camera::create(perspective, lookAt,
    vsg::ViewportState::create(window->extent2D()));
```

Essential Event Handlers

```
// Close window on ESC or window close button
viewer->addEventHandler(vsg::CloseHandler::create(viewer));

// Mouse camera control
viewer->addEventHandler(vsg::Trackball::create(camera));

// Keyboard input
class KeyHandler : public vsg::Inherit<vsg::Visitor, KeyHandler>
{
public:
    void apply(vsg::KeyPressEvent& keyPress) override
    {
        if (keyPress.keyBase == 'f')
        {
            // Toggle fullscreen
        }
    }
};
viewer->addEventHandler(KeyHandler::create());
```

File Loading with vsgXchange

```
// Create options with all format support
auto options = vsg::Options::create(vsgXchange::all::create());

// Set search paths from environment
```

```
options->paths = vsg::getEnvPaths("VSG_FILE_PATH");

// Enable file caching
options->fileCache = vsg::getEnv("VSG_FILE_CACHE");

// Load with type checking
if (auto node = vsg::read_cast<vsg::Node>(filename, options))
{
    // Successfully loaded as Node
}
```

Cameras

VSG provides a flexible camera system supporting various projection types and view configurations.

Camera Components

A camera consists of three main components:

```
// View matrix - defines camera position and orientation
auto lookAt = vsg::LookAt::create(
    eye,    // Camera position
    center, // Look at point
    up      // Up vector
);

// Projection matrix - defines the projection
auto perspective = vsg::Perspective::create(
    fieldOfViewY,  // Vertical field of view in degrees
    aspectRatio,   // Width/height ratio
    nearDistance,  // Near clipping plane
    farDistance    // Far clipping plane
);

// Viewport - defines the rendering area
auto viewport = vsg::ViewportState::create(x, y, width, height);

// Combine into camera
auto camera = vsg::Camera::create(perspective, lookAt, viewport);
```

Camera Types

Standard Cameras

```
// Perspective projection
auto perspective = vsg::Perspective::create(
    30.0,                              // FOV
    aspectRatio,                       // Aspect
    0.1,                              // Near
    1000.0                            // Far
);

// Orthographic projection
auto ortho = vsg::Orthographic::create(
    -1.0, 1.0,  // Left, right
    -1.0, 1.0,  // Bottom, top
    0.1, 100.0  // Near, far
);

// Relative projection (based on bounding sphere)
auto relative = vsg::RelativeProjection::create(
    sphere,      // Bounding sphere
    fieldOfView, // FOV
    aspectRatio  // Aspect
);
```

Specialized Cameras

```
// For Earth/planetary rendering
auto ellipsoidPerspective = vsg::EllipsoidPerspective::create(
```

```
    lookAt,
    ellipsoidModel,
    fieldOfViewY,
    aspectRatio,
    nearFarRatio,
    horizonMountainHeight
);

// Dynamic view matrix that tracks scene graph nodes
auto trackingView = vsg::TrackingViewMatrix::create(nodePath);
```

Multiple Views

```
// Create multiple viewports in one window
auto commandGraph = vsg::CommandGraph::create(window);

// Main view - full window
auto mainView = vsg::View::create(mainCamera, scene);
auto mainRenderGraph = vsg::RenderGraph::create(window, mainView);
commandGraph->addChild(mainRenderGraph);

// Secondary view - picture-in-picture
auto pipViewport = vsg::ViewportState::create(10, 10, 320, 240);
auto pipCamera = vsg::Camera::create(perspective, lookAt, pipViewport);
auto pipView = vsg::View::create(pipCamera, scene);
auto pipRenderGraph = vsg::RenderGraph::create(window, pipView);
commandGraph->addChild(pipRenderGraph);
```

Camera Animation

```
// Load camera animation path
auto cameraAnimation = vsg::CameraAnimationHandler::create(
    camera,
    pathFilename,
    options
);
viewer->addEventHandler(cameraAnimation);
cameraAnimation->play();

// Programmatic camera animation
class AnimateCamera : public vsg::Visitor
{
    vsg::ref_ptr<vsg::Camera> camera;

    void apply(vsg::FrameEvent& frame) override
    {
        double time = frame.frameStamp->simulationTime;

        auto lookAt = camera->viewMatrix.cast<vsg::LookAt>();
        if (lookAt)
        {
            // Orbit camera around center
            double angle = time * 0.5;
            lookAt->eye = lookAt->center +
                vsg::dvec3(cos(angle) * radius, sin(angle) * radius, height);
        }
    }
};
```

Finding and Managing Cameras

```
// Find all cameras in a scene
auto cameras = vsg::visit<vsg::FindCameras>(scene).cameras;

for (auto& [nodePath, camera] : cameras)
{
    std::cout << "Found camera: " << camera->name << std::endl;

    // Get the world transform of the camera
    auto worldTransform = vsg::visit<vsg::ComputeTransform>(
```

```
        nodePath.begin(), nodePath.end()
    ).matrix;
}

// Name cameras for identification
camera->name = "MainCamera";
camera->setValue("type", "perspective");
```

Viewports and Windows

VSG provides flexible window management supporting single and multi-window applications.

Window Creation

```
// Basic window
auto traits = vsg::WindowTraits::create();
traits->windowTitle = "My Application";
traits->width = 1280;
traits->height = 720;
traits->fullscreen = false;
traits->samples = VK_SAMPLE_COUNT_8_BIT;
traits->debugLayer = true;

auto window = vsg::Window::create(traits);

// Add to viewer
auto viewer = vsg::Viewer::create();
viewer->addWindow(window);
```

Window Traits Configuration

```
auto traits = vsg::WindowTraits::create();

// Basic properties
traits->windowTitle = "VSG Window";
traits->x = 100;           // Window position
traits->y = 100;
traits->width = 1920;
traits->height = 1080;
traits->fullscreen = false;
traits->decoration = true;  // Window frame
traits->screenNum = 0;     // Monitor selection

// Vulkan configuration
traits->samples = VK_SAMPLE_COUNT_4_BIT;    // MSAA
traits->debugLayer = true;                   // Validation
traits->apiDumpLayer = false;               // API logging
traits->synchronizationLayer = false;       // Sync validation

// Swapchain preferences
traits->swapchainPreferences.imageCount = 3;             // Triple buffering
traits->swapchainPreferences.presentMode = VK_PRESENT_MODE_FIFO_KHR;
traits->swapchainPreferences.surfaceFormat = {
    VK_FORMAT_B8G8R8A8_SRGB,
    VK_COLOR_SPACE_SRGB_NONLINEAR_KHR
};

// Advanced features
traits->depthFormat = VK_FORMAT_D32_SFLOAT;
traits->depthImageUsage = VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT;
```

Multiple Windows

```
// Create multiple windows
auto viewer = vsg::Viewer::create();

// First window
auto window1 = vsg::Window::create(traits1);
viewer->addWindow(window1);
```

```
// Second window - can share device
auto traits2 = vsg::WindowTraits::create(*traits1);
traits2->windowTitle = "Second Window";
traits2->device = window1->getOrCreateDevice();  // Share device
auto window2 = vsg::Window::create(traits2);
viewer->addWindow(window2);

// Each window gets its own command graph
auto commandGraph1 = vsg::CommandGraph::create(window1);
auto commandGraph2 = vsg::CommandGraph::create(window2);

viewer->assignRecordAndSubmitTaskAndPresentation({
    commandGraph1,
    commandGraph2
});
```

Viewports

```
// Full window viewport (default)
auto viewport = vsg::ViewportState::create(
    0, 0,                              // x, y
    window->extent2D().width,          // width
    window->extent2D().height          // height
);

// Multiple viewports in one window
// Top half
auto topViewport = vsg::ViewportState::create(
    0, 0,
    window->extent2D().width,
    window->extent2D().height / 2
);

// Bottom half
auto bottomViewport = vsg::ViewportState::create(
    0, window->extent2D().height / 2,
    window->extent2D().width,
    window->extent2D().height / 2
);

// Create cameras with different viewports
auto topCamera = vsg::Camera::create(projection, view, topViewport);
auto bottomCamera = vsg::Camera::create(projection, view, bottomViewport);
```

Window Events

```
// Window-specific event handling
class WindowEventHandler : public vsg::Inherit<vsg::Visitor, WindowEventHandler>
{
    vsg::ref_ptr<vsg::Window> window;

public:
    void apply(vsg::ConfigureWindowEvent& event) override
    {
        // Window resized or moved
        auto& extent = event.window->extent2D();
        std::cout << "Window resized to " << extent.width
                  << "x" << extent.height << std::endl;
    }

    void apply(vsg::ExposeWindowEvent& event) override
    {
        // Window needs redraw
    }

    void apply(vsg::CloseWindowEvent& event) override
    {
        // Window close requested
        event.window->close();
    }
```

```
};

// Restrict handler to specific window
auto handler = WindowEventHandler::create();
handler->addWindow(window1);  // Only responds to window1 events
viewer->addEventHandler(handler);
```

Render Configuration

```
// Create render graph with custom clear values
auto renderGraph = vsg::RenderGraph::create(window);

// Set clear color (linear color space)
renderGraph->clearValues[0].color = {{0.2f, 0.2f, 0.3f, 1.0f}};

// Convert from sRGB if needed
renderGraph->clearValues[0].color =
    vsg::sRGB_to_linear(0.2f, 0.2f, 0.3f, 1.0f);

// Add render graph to command graph
auto commandGraph = vsg::CommandGraph::create(window);
commandGraph->addChild(renderGraph);
```

Dynamic Window Management

```
// Add/remove windows at runtime
viewer->addWindow(newWindow);
viewer->removeWindow(oldWindow);

// Query windows
for (auto& window : viewer->windows())
{
    std::cout << "Window: " << window->traits()->windowTitle
              << " [" << window->extent2D().width
              << "x" << window->extent2D().height << "]" << std::endl;
}

// Check if window is valid
if (window->valid())
{
    // Window is open and functional
}
```

Render to Texture

Render to texture (RTT) is a fundamental technique for many advanced rendering effects.

Basic Setup

```
// Create render target image
auto colorImage = vsg::Image::create();
colorImage->imageType = VK_IMAGE_TYPE_2D;
colorImage->format = VK_FORMAT_R8G8B8A8_SRGB;
colorImage->extent = {width, height, 1};
colorImage->mipLevels = 1;
colorImage->arrayLayers = 1;
colorImage->samples = VK_SAMPLE_COUNT_1_BIT;
colorImage->tiling = VK_IMAGE_TILING_OPTIMAL;
colorImage->usage = VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT |
                   VK_IMAGE_USAGE_SAMPLED_BIT;  // Both render and sample
colorImage->initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;

// Create image view
auto colorImageView = vsg::createImageView(context, colorImage,
                                          VK_IMAGE_ASPECT_COLOR_BIT);

// Create depth buffer (if needed)
auto depthImage = vsg::Image::create();
depthImage->format = VK_FORMAT_D32_SFLOAT;
depthImage->extent = {width, height, 1};
```

```
depthImage->usage = VK_IMAGE_USAGE_DEPTH_STENCIL_ATTACHMENT_BIT;
auto depthImageView = vsg::createImageView(context, depthImage,
                                           VK_IMAGE_ASPECT_DEPTH_BIT);
```

Render Pass Configuration

```
// Attachment descriptions
vsg::RenderPass::Attachments attachments(2);

// Color attachment
attachments[0].format = VK_FORMAT_R8G8B8A8_SRGB;
attachments[0].samples = VK_SAMPLE_COUNT_1_BIT;
attachments[0].loadOp = VK_ATTACHMENT_LOAD_OP_CLEAR;
attachments[0].storeOp = VK_ATTACHMENT_STORE_OP_STORE;
attachments[0].stencilLoadOp = VK_ATTACHMENT_LOAD_OP_DONT_CARE;
attachments[0].stencilStoreOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
attachments[0].initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
attachments[0].finalLayout = VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL;

// Depth attachment
attachments[1].format = VK_FORMAT_D32_SFLOAT;
attachments[1].samples = VK_SAMPLE_COUNT_1_BIT;
attachments[1].loadOp = VK_ATTACHMENT_LOAD_OP_CLEAR;
attachments[1].storeOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
attachments[1].initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
attachments[1].finalLayout = VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL;

// Subpass
vsg::AttachmentReference colorRef = {0, VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL};
vsg::AttachmentReference depthRef = {1, VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL};

vsg::RenderPass::Subpasses subpasses(1);
subpasses[0].pipelineBindPoint = VK_PIPELINE_BIND_POINT_GRAPHICS;
subpasses[0].colorAttachments = {colorRef};
subpasses[0].depthStencilAttachments = {depthRef};
```

Critical: Subpass Dependencies

```
vsg::RenderPass::Dependencies dependencies(2);

// Transition from previous usage to color attachment
dependencies[0].srcSubpass = VK_SUBPASS_EXTERNAL;
dependencies[0].dstSubpass = 0;
dependencies[0].srcStageMask = VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT;
dependencies[0].dstStageMask = VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT;
dependencies[0].srcAccessMask = VK_ACCESS_SHADER_READ_BIT;
dependencies[0].dstAccessMask = VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT;
dependencies[0].dependencyFlags = VK_DEPENDENCY_BY_REGION_BIT;

// Transition from color attachment to shader read
dependencies[1].srcSubpass = 0;
dependencies[1].dstSubpass = VK_SUBPASS_EXTERNAL;
dependencies[1].srcStageMask = VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT;
dependencies[1].dstStageMask = VK_PIPELINE_STAGE_FRAGMENT_SHADER_BIT;
dependencies[1].srcAccessMask = VK_ACCESS_COLOR_ATTACHMENT_WRITE_BIT;
dependencies[1].dstAccessMask = VK_ACCESS_SHADER_READ_BIT;
dependencies[1].dependencyFlags = VK_DEPENDENCY_BY_REGION_BIT;

// Create render pass
auto renderPass = vsg::RenderPass::create(device, attachments, subpasses, dependencies);
```

Framebuffer and RenderGraph

```
// Create framebuffer
auto framebuffer = vsg::Framebuffer::create(
    renderPass,
    vsg::ImageViews{colorImageView, depthImageView},
    width, height, 1
);
```

```
// Create offscreen render graph
auto offscreenGraph = vsg::RenderGraph::create();
offscreenGraph->framebuffer = framebuffer;
offscreenGraph->renderPass = renderPass;
offscreenGraph->renderArea = {{0, 0}, {width, height}};
offscreenGraph->clearValues = {
    VkClearValue{{{0.2f, 0.2f, 0.4f, 1.0f}}},  // Clear color
    VkClearValue{{{1.0f, 0}}}                   // Clear depth
};

// Add scene to render
auto view = vsg::View::create(camera, scene);
offscreenGraph->addChild(view);
```

Using the Rendered Texture

```
// Create sampler for texture access
auto sampler = vsg::Sampler::create();
sampler->magFilter = VK_FILTER_LINEAR;
sampler->minFilter = VK_FILTER_LINEAR;
sampler->addressModeU = VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE;
sampler->addressModeV = VK_SAMPLER_ADDRESS_MODE_CLAMP_TO_EDGE;

// Create descriptor for shader access
auto textureInfo = vsg::ImageInfo::create(
    sampler,
    colorImageView,
    VK_IMAGE_LAYOUT_SHADER_READ_ONLY_OPTIMAL
);

auto texture = vsg::DescriptorImage::create(
    textureInfo,
    0,  // binding
    0,  // array element
    VK_DESCRIPTOR_TYPE_COMBINED_IMAGE_SAMPLER
);

// Use in material/state
auto descriptorSet = vsg::DescriptorSet::create(
    descriptorSetLayout,
    vsg::Descriptors{texture}
);
```

Complete Command Graph

```
// Command graph with both passes
auto commandGraph = vsg::CommandGraph::create(window);

// First: Render to texture
commandGraph->addChild(offscreenGraph);

// Second: Render to screen using texture
auto mainGraph = vsg::RenderGraph::create(window);
mainGraph->addChild(mainView);
commandGraph->addChild(mainGraph);

viewer->assignRecordAndSubmitTaskAndPresentation({commandGraph});
```

Offscreen Rendering and Image Capture

VSG provides sophisticated support for offscreen rendering and capturing rendered images to files, enabling high-quality screenshot generation and batch rendering workflows.

Image Transfer Operations

```
// Check if device supports format blitting
bool supportsBlit(vsg::ref_ptr<vsg::Device> device, VkFormat sourceFormat)
{
    auto physicalDevice = device->getPhysicalDevice();
    VkFormatProperties srcProps, dstProps;
```

```
    vkGetPhysicalDeviceFormatProperties(*physicalDevice, sourceFormat, &srcProps);
    vkGetPhysicalDeviceFormatProperties(*physicalDevice, VK_FORMAT_R8G8B8A8_SRGB, &dstProps);

    return ((srcProps.optimalTilingFeatures & VK_FORMAT_FEATURE_BLIT_SRC_BIT) != 0) &&
           ((dstProps.linearTilingFeatures & VK_FORMAT_FEATURE_BLIT_DST_BIT) != 0);
}

// Create capture image with host-visible memory
auto captureImage = vsg::Image::create();
captureImage->format = targetFormat;
captureImage->extent = {width, height, 1};
captureImage->tiling = VK_IMAGE_TILING_LINEAR;  // Essential for CPU access
captureImage->usage = VK_IMAGE_USAGE_TRANSFER_DST_BIT;

// Allocate host-visible, coherent memory
auto memReqs = captureImage->getMemoryRequirements(device->deviceID);
auto memFlags = VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT | VK_MEMORY_PROPERTY_HOST_COHERENT_BIT;
auto deviceMemory = vsg::DeviceMemory::create(device, memReqs, memFlags);
captureImage->bind(deviceMemory, 0);
```

Transfer Commands with Proper Synchronization

```
auto transferCommands = vsg::Commands::create();

// Transition destination to transfer layout
auto transitionBarrier = vsg::ImageMemoryBarrier::create(
    0,                                        // srcAccessMask
    VK_ACCESS_TRANSFER_WRITE_BIT,            // dstAccessMask
    VK_IMAGE_LAYOUT_UNDEFINED,               // oldLayout
    VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL,    // newLayout
    VK_QUEUE_FAMILY_IGNORED,
    VK_QUEUE_FAMILY_IGNORED,
    destinationImage,
    VkImageSubresourceRange{VK_IMAGE_ASPECT_COLOR_BIT, 0, 1, 0, 1}
);

transferCommands->addChild(vsg::PipelineBarrier::create(
    VK_PIPELINE_STAGE_TRANSFER_BIT,
    VK_PIPELINE_STAGE_TRANSFER_BIT,
    0, transitionBarrier));

// Choose copy or blit based on format compatibility
if (sameFormatAndSize)
{
    // Direct copy
    VkImageCopy region{};
    region.srcSubresource = {VK_IMAGE_ASPECT_COLOR_BIT, 0, 0, 1};
    region.dstSubresource = {VK_IMAGE_ASPECT_COLOR_BIT, 0, 0, 1};
    region.extent = destinationImage->extent;

    auto copyImage = vsg::CopyImage::create();
    copyImage->srcImage = sourceImage;
    copyImage->dstImage = destinationImage;
    copyImage->srcImageLayout = VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL;
    copyImage->dstImageLayout = VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL;
    copyImage->regions.push_back(region);
    transferCommands->addChild(copyImage);
}
else
{
    // Blit for format/size conversion
    VkImageBlit region{};
    region.srcSubresource = {VK_IMAGE_ASPECT_COLOR_BIT, 0, 0, 1};
    region.dstSubresource = {VK_IMAGE_ASPECT_COLOR_BIT, 0, 0, 1};
    region.srcOffsets[1] = {static_cast<int32_t>(sourceImage->extent.width),
                            static_cast<int32_t>(sourceImage->extent.height), 1};
    region.dstOffsets[1] = {static_cast<int32_t>(destinationImage->extent.width),
                            static_cast<int32_t>(destinationImage->extent.height), 1};

    auto blitImage = vsg::BlitImage::create();
    blitImage->srcImage = sourceImage;
```

```
    blitImage->dstImage = destinationImage;
    blitImage->srcImageLayout = VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL;
    blitImage->dstImageLayout = VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL;
    blitImage->regions.push_back(region);
    blitImage->filter = VK_FILTER_NEAREST;
    transferCommands->addChild(blitImage);
}

// Transition to general layout for CPU access
auto finalTransition = vsg::ImageMemoryBarrier::create(
    VK_ACCESS_TRANSFER_WRITE_BIT,            // srcAccessMask
    VK_ACCESS_MEMORY_READ_BIT,               // dstAccessMask
    VK_IMAGE_LAYOUT_TRANSFER_DST_OPTIMAL,    // oldLayout
    VK_IMAGE_LAYOUT_GENERAL,                 // newLayout
    VK_QUEUE_FAMILY_IGNORED,
    VK_QUEUE_FAMILY_IGNORED,
    destinationImage,
    VkImageSubresourceRange{VK_IMAGE_ASPECT_COLOR_BIT, 0, 1, 0, 1}
);

transferCommands->addChild(vsg::PipelineBarrier::create(
    VK_PIPELINE_STAGE_TRANSFER_BIT,
    VK_PIPELINE_STAGE_TRANSFER_BIT,
    0, finalTransition));
```

Custom Render Pass for Transfer

```
vsg::ref_ptr<vsg::RenderPass> createTransferRenderPass(
    vsg::ref_ptr<vsg::Device> device,
    VkFormat imageFormat,
    VkFormat depthFormat)
{
    auto colorAttachment = vsg::defaultColorAttachment(imageFormat);
    colorAttachment.finalLayout = VK_IMAGE_LAYOUT_TRANSFER_SRC_OPTIMAL; // Key difference

    auto depthAttachment = vsg::defaultDepthAttachment(depthFormat);

    vsg::RenderPass::Attachments attachments{colorAttachment, depthAttachment};
    // ... configure subpasses and dependencies

    return vsg::RenderPass::create(device, attachments, subpasses, dependencies);
}
```

Accessing Image Data from GPU Memory

```
// Wait for GPU operations to complete
viewer->waitForFences(0, 1000000000); // 1 second timeout

// Get memory layout information
VkImageSubresource subResource{VK_IMAGE_ASPECT_COLOR_BIT, 0, 0};
VkSubresourceLayout subResourceLayout;
vkGetImageSubresourceLayout(*device, captureImage->vk(device->deviceID),
                           &subResource, &subResourceLayout);

auto deviceMemory = captureImage->getDeviceMemory(device->deviceID);
size_t destRowWidth = captureImage->extent.width * sizeof(vsg::ubvec4);

if (destRowWidth == subResourceLayout.rowPitch)
{
    // Contiguous memory - direct mapping
    auto imageData = vsg::MappedData<vsg::ubvec4Array2D>::create(
        deviceMemory,
        subResourceLayout.offset,
        0,
        vsg::Data::Properties{captureImage->format},
        captureImage->extent.width,
        captureImage->extent.height
    );
    return imageData;
}
```

```cpp
else
{
    // Non-contiguous - copy row by row
    auto mappedData = vsg::MappedData<vsg::ubyteArray>::create(
        deviceMemory, subResourceLayout.offset, 0,
        vsg::Data::Properties{captureImage->format},
        subResourceLayout.rowPitch * captureImage->extent.height
    );

    auto imageData = vsg::ubvec4Array2D::create(
        captureImage->extent.width, captureImage->extent.height,
        vsg::Data::Properties{captureImage->format}
    );

    for (uint32_t row = 0; row < captureImage->extent.height; ++row)
    {
        std::memcpy(
            imageData->dataPointer(row * captureImage->extent.width),
            mappedData->dataPointer(row * subResourceLayout.rowPitch),
            destRowWidth
        );
    }
    return imageData;
}
```

Complete Offscreen Rendering Pipeline

```cpp
// Setup offscreen rendering
auto offscreenCommandGraph = vsg::CommandGraph::create(window);
offscreenCommandGraph->submitOrder = -1; // Render before display

// Create transfer image view for offscreen rendering
auto transferImageView = createTransferImageView(device, format, extent, samples);
auto captureImage = createCaptureImage(device, format, extent);
auto transferCommands = createTransferCommands(device, transferImageView->image, captureImage);

// Setup offscreen framebuffer and render graph
auto framebuffer = createOffscreenFramebuffer(device, transferImageView, samples);
auto offscreenRenderGraph = vsg::RenderGraph::create();
offscreenRenderGraph->framebuffer = framebuffer;
offscreenRenderGraph->renderArea.extent = extent;

// Configure camera with independent viewport
auto offscreenCamera = vsg::Camera::create();
offscreenCamera->viewMatrix = displayCamera->viewMatrix;        // Share view
offscreenCamera->projectionMatrix = independentProjection;      // Independent aspect
offscreenCamera->viewportState = vsg::ViewportState::create(extent);

// Add to command graph with switch for conditional rendering
auto offscreenSwitch = vsg::Switch::create();
offscreenSwitch->addChild(false, offscreenRenderGraph);  // Initially disabled
offscreenSwitch->addChild(false, transferCommands);
offscreenCommandGraph->addChild(offscreenSwitch);

// Enable offscreen rendering when needed
offscreenSwitch->setAllChildren(true);
```

Key Concepts

- **Memory Layout**: GPU memory may not be contiguous; handle row pitch correctly
- **Format Support**: Check device capabilities for format conversion and blitting
- **Synchronization**: Use proper barriers and wait for GPU completion
- **Transfer Layout**: Images must be in TRANSFER_SRC_OPTIMAL for reading
- **Host Memory**: Use VK_MEMORY_PROPERTY_HOST_VISIBLE_BIT for CPU access
- **Resolution Independence**: Offscreen rendering can use arbitrary resolutions

State Management

Graphics Pipeline

Content will be added as examples are documented

Shaders

Content will be added as examples are documented

Subpasses and Render Passes

VSG provides powerful support for Vulkan subpasses, enabling efficient multi-pass rendering within a single render pass. This keeps intermediate results in tile memory rather than writing to main memory, providing significant performance benefits especially on mobile GPUs.

Multi-Subpass Render Pass Creation

```
vsg::ref_ptr<vsg::RenderPass> createMultiSubpassRenderPass(vsg::Device* device)
{
    // Define attachments
    vsg::AttachmentDescription colorAttachment = {};
    colorAttachment.format = VK_FORMAT_B8G8R8A8_UNORM;
    colorAttachment.samples = VK_SAMPLE_COUNT_1_BIT;
    colorAttachment.loadOp = VK_ATTACHMENT_LOAD_OP_CLEAR;
    colorAttachment.storeOp = VK_ATTACHMENT_STORE_OP_STORE;
    colorAttachment.initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
    colorAttachment.finalLayout = VK_IMAGE_LAYOUT_PRESENT_SRC_KHR;

    vsg::AttachmentDescription depthAttachment = {};
    depthAttachment.format = VK_FORMAT_D24_UNORM_S8_UINT;
    depthAttachment.loadOp = VK_ATTACHMENT_LOAD_OP_CLEAR;
    depthAttachment.storeOp = VK_ATTACHMENT_STORE_OP_DONT_CARE;
    depthAttachment.initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;
    depthAttachment.finalLayout = VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL;

    vsg::RenderPass::Attachments attachments{colorAttachment, depthAttachment};

    // Define attachment references
    vsg::AttachmentReference colorRef = {0, VK_IMAGE_LAYOUT_COLOR_ATTACHMENT_OPTIMAL};
    vsg::AttachmentReference depthRef = {1, VK_IMAGE_LAYOUT_DEPTH_STENCIL_ATTACHMENT_OPTIMAL};

    // First subpass - with depth testing
    vsg::SubpassDescription subpass1;
    subpass1.pipelineBindPoint = VK_PIPELINE_BIND_POINT_GRAPHICS;
    subpass1.colorAttachments.emplace_back(colorRef);
    subpass1.depthStencilAttachments.emplace_back(depthRef);

    // Second subpass - without depth testing
    vsg::SubpassDescription subpass2;
    subpass2.pipelineBindPoint = VK_PIPELINE_BIND_POINT_GRAPHICS;
    subpass2.colorAttachments.emplace_back(colorRef);
    // No depth attachment = no depth testing

    vsg::RenderPass::Subpasses subpasses{subpass1, subpass2};

    // Define dependencies between subpasses
    vsg::SubpassDependency dependency = {};
    dependency.srcSubpass = 0;  // First subpass
    dependency.dstSubpass = 1;  // Second subpass
    dependency.srcStageMask = VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT;
    dependency.dstStageMask = VK_PIPELINE_STAGE_COLOR_ATTACHMENT_OUTPUT_BIT;
    dependency.srcAccessMask = VK_ACCESS_COLOR_ATTACHMENT_READ_BIT;
    dependency.dstAccessMask = 0;

    vsg::RenderPass::Dependencies dependencies{dependency};

    return vsg::RenderPass::create(device, attachments, subpasses, dependencies);
}
```

Subpass-Specific Pipeline Creation

```
// Create pipelines targeting specific subpasses
auto createPipelineForSubpass(uint32_t subpassIndex,
                              vsg::ref_ptr<vsg::PipelineLayout> layout,
                              const vsg::ShaderStages& shaders,
```

```cpp
                             const vsg::GraphicsPipelineStates& states)
{
    return vsg::GraphicsPipeline::create(
        layout,
        shaders,
        states,
        subpassIndex  // Key: specify target subpass
    );
}

// Pipeline for first subpass (index 0)
auto pipeline1 = createPipelineForSubpass(0, pipelineLayout, shaders, states);

// Pipeline for second subpass (index 1)
auto pipeline2 = createPipelineForSubpass(1, pipelineLayout, shaders, states);
```

Scene Graph with Subpass Transitions

```cpp
// Organize scene graph for multi-subpass rendering
auto scenegraph = vsg::StateGroup::create();

// First subpass content
auto subpass1Content = vsg::StateGroup::create();
subpass1Content->add(vsg::BindGraphicsPipeline::create(pipeline1));
subpass1Content->add(vsg::BindDescriptorSets::create(/* descriptors */));
subpass1Content->addChild(/* geometry commands */);

// Second subpass content
auto subpass2Content = vsg::StateGroup::create();
subpass2Content->add(vsg::BindGraphicsPipeline::create(pipeline2));
subpass2Content->add(vsg::BindDescriptorSets::create(/* descriptors */));
subpass2Content->addChild(/* geometry commands */);

// Combine with subpass transition
scenegraph->addChild(subpass1Content);
scenegraph->addChild(vsg::NextSubPass::create(VK_SUBPASS_CONTENTS_INLINE));
scenegraph->addChild(subpass2Content);
```

Setting Custom Render Pass

```cpp
// Apply custom render pass to window
auto window = vsg::Window::create(windowTraits);
auto customRenderPass = createMultiSubpassRenderPass(window->getOrCreateDevice());
window->setRenderPass(customRenderPass);
```

Common Subpass Patterns

Deferred Rendering: `cpp // Subpass 1: G-buffer generation // Subpass 2: Lighting calculation using G-buffer data // Subpass 3: Forward rendering for transparent objects`

UI Overlay: `cpp // Subpass 1: 3D scene rendering with depth testing // Subpass 2: UI overlay rendering without depth testing`

Post-Processing: `cpp // Subpass 1: Scene rendering to intermediate targets // Subpass 2: Post-processing effects using intermediate results`

Key Benefits

- **Memory Bandwidth**: Intermediate results stay in tile memory
- **Performance**: Eliminates expensive memory round-trips
- **Mobile Optimization**: Essential for tile-based renderers
- **Automatic Synchronization**: Dependencies handled by render pass
- **Resource Efficiency**: Shared attachments between subpasses

Dynamic State

Content will be added as examples are documented

Textures

VSG provides comprehensive support for various texture types including 2D texture arrays, which enable efficient multi-target rendering scenarios.

2D Texture Arrays

Texture arrays allow multiple textures of identical format and size to be stored in a single image object, providing significant performance benefits.

```
// Create 2D texture array
vsg::ref_ptr<vsg::Image> createTextureArray(vsg::Context& context,
                                           uint32_t width, uint32_t height,
                                           uint32_t layers, VkFormat format)
{
    auto image = vsg::Image::create();
    image->imageType = VK_IMAGE_TYPE_2D;
    image->format = format;
    image->extent = VkExtent3D{width, height, 1};
    image->mipLevels = 1;
    image->arrayLayers = layers;  // Multiple layers
    image->samples = VK_SAMPLE_COUNT_1_BIT;
    image->tiling = VK_IMAGE_TILING_OPTIMAL;
    image->usage = VK_IMAGE_USAGE_COLOR_ATTACHMENT_BIT | VK_IMAGE_USAGE_SAMPLED_BIT;
    image->initialLayout = VK_IMAGE_LAYOUT_UNDEFINED;

    image->compile(context);
    return image;
}
```

Layer-Specific Image Views

Access individual layers of texture arrays through specialized image views:

```
// Create image view for specific array layer
auto createLayerImageView(vsg::ref_ptr<vsg::Image> arrayImage, uint32_t layer)
{
    auto imageView = vsg::ImageView::create(arrayImage, VK_IMAGE_ASPECT_COLOR_BIT);
    imageView->subresourceRange.baseArrayLayer = layer;  // Target specific layer
    imageView->subresourceRange.layerCount = 1;          // Single layer view
    imageView->compile(context);
    return imageView;
}

// For rendering to specific layer
auto colorImageView = createLayerImageView(textureArray, targetLayer);

// For sampling from specific layer
auto samplerImageView = createLayerImageView(textureArray, sourceLayer);
```

Render to Texture Array Layers

Configure framebuffers to target individual array layers:

```
// Create framebuffer targeting specific layer
auto createLayerFramebuffer(vsg::ref_ptr<vsg::Image> colorArray,
                           vsg::ref_ptr<vsg::Image> depthArray,
                           uint32_t layer, VkExtent2D extent)
{
    auto colorView = createLayerImageView(colorArray, layer);
    auto depthView = createLayerImageView(depthArray, layer);

    // Render pass configured for texture array usage
    auto renderPass = createTextureArrayRenderPass(colorArray->format, depthArray->format);

    auto framebuffer = vsg::Framebuffer::create(
        renderPass,
        vsg::ImageViews{colorView, depthView},
        extent.width, extent.height, 1
    );

    return framebuffer;
}
```

Texture Array Sampling in Shaders

Access texture arrays in fragment shaders using array indexing:

```
// Fragment shader for texture array sampling
layout(binding = 0) uniform sampler2DArray textureArray;

layout(location = 0) in vec2 texCoord;
layout(location = 1) flat in int arrayIndex;  // Layer index
layout(location = 0) out vec4 outColor;

void main()
{
    // Sample from specific array layer
    outColor = texture(textureArray, vec3(texCoord, float(arrayIndex)));
}
```

Performance Optimization with Shared Views

Reduce pipeline duplication by sharing view IDs across similar render passes:

```
// Efficient multi-layer rendering
vsg::ref_ptr<vsg::View> firstView;
for (uint32_t layer = 0; layer < numLayers; ++layer)
{
    auto camera = createCameraForLayer(layer);

    vsg::ref_ptr<vsg::View> view;
    if (shareViewID && firstView)
    {
        view = vsg::View::create(*firstView);  // Copy view (same viewID)
        view->camera = camera;                  // Different camera
    }
    else
    {
        view = vsg::View::create(camera, scene);
        if (!firstView) firstView = view;
    }

    renderGraph->addChild(view);
}
```

Common Use Cases

Shadow Mapping: ```cpp // Create shadow map array for multiple lights auto shadowMapArray = createTextureArray(context, 1024, 1024, numLights, VK_FORMAT_D32_SFLOAT);

// Render shadow maps for each light for (uint32_t light = 0; light < numLights; ++light) { auto lightCamera = createLightCamera(lights[light]); auto shadowFramebuffer = createLayerFramebuffer(shadowMapArray, nullptr, light, {1024, 1024}); // ... render shadow map } ```

Reflection Probes: ```cpp // Cube map faces as array layers auto reflectionArray = createTextureArray(context, 256, 256, 6, VK_FORMAT_R8G8B8A8_SRGB);

// Render each cube face for (uint32_t face = 0; face < 6; ++face) { auto faceCamera = createCubeFaceCamera(probePosition, face); auto faceFramebuffer = createLayerFramebuffer(reflectionArray, depthArray, face, {256, 256}); // ... render cube face } ```

Key Benefits

- **Memory Efficiency**: Single allocation for multiple textures
- **Reduced Binding Overhead**: One descriptor set for entire array
- **Pipeline Optimization**: Shared view IDs reduce pipeline objects
- **Shader Flexibility**: Dynamic array indexing in shaders
- **Consistent Properties**: All layers have identical format and size

Color Space Management

VSG provides robust color space handling for proper color reproduction across different display devices and formats.

Color Space Conversion Functions

```
// Convert sRGB values to linear space
float linear_value = vsg::sRGB_to_linear(0.5f);

// Convert linear values to sRGB space
float srgb_value = vsg::linear_to_sRGB(0.5f);

// Format conversion utilities
VkFormat linear_format = vsg::sRGB_to_uNorm(srgb_format);
VkFormat srgb_format = vsg::uNorm_to_sRGB(linear_format);
```

Surface Format Enumeration

```
// Query supported surface formats
auto physicalDevice = window->getOrCreatePhysicalDevice();
auto surface = window->getOrCreateSurface();
auto swapChainSupportDetails = vsg::querySwapChainSupport(physicalDevice->vk(), surface->vk());

// Create windows for different formats
for(auto& format : swapChainSupportDetails.formats)
{
    auto windowTraits = vsg::WindowTraits::create();
    windowTraits->swapchainPreferences.surfaceFormat = format;
    auto window = vsg::Window::create(windowTraits);
}
```

Image Format Conversion

```
// Load image and convert formats
auto image = vsg::read_cast<vsg::Data>("texture.jpg", options);

// Convert to linear format
auto image_linear = vsg::clone(image);
image_linear->properties.format = vsg::sRGB_to_uNorm(image->properties.format);

// Convert to sRGB format
auto image_sRGB = vsg::clone(image);
image_sRGB->properties.format = vsg::uNorm_to_sRGB(image->properties.format);
```

Common Surface Formats

- **VK_FORMAT_B8G8R8A8_SRGB** - 8-bit sRGB with alpha
- **VK_FORMAT_B8G8R8A8_UNORM** - 8-bit linear with alpha
- **VK_FORMAT_R8G8B8A8_SRGB** - 8-bit sRGB, different channel order
- **VK_FORMAT_A2B10G10R10_UNORM_PACK32** - 10-bit linear, high precision

Color Space Best Practices

- Use sRGB formats for final display output
- Use linear formats for intermediate calculations
- Convert textures to appropriate format based on usage
- Be aware of gamma correction implications
- Test with different surface formats for compatibility

Stereo Rendering

VSG provides excellent support for stereo rendering through multiple views and relative cameras. This enables stereoscopic 3D effects and virtual reality applications.

Anaglyphic Stereo (Red/Cyan 3D)

Anaglyphic stereo renders two views with different color masks to create a stereoscopic effect viewable with red/cyan 3D glasses.

Stereo Camera Setup

```
// Master camera parameters
double eyeSeperation = 0.06;      // 6cm human eye separation
double screenDistance = 0.75;     // 75cm viewing distance
double screenWidth = 0.55;        // Physical screen width
double shear = (eyeSeperation / screenWidth) * 0.8;
```

```
// Create master camera
auto lookAt = vsg::LookAt::create(eye, center, up);
auto perspective = vsg::Perspective::create(fov, aspect, near, far);
auto master_camera = vsg::Camera::create(perspective, lookAt, viewport);

// Left eye camera - offset left, renders to red channel
auto left_relative_perspective = vsg::RelativeProjection::create(
    vsg::translate(-shear, 0.0, 0.0), perspective);
auto left_relative_view = vsg::RelativeViewMatrix::create(
    vsg::translate(-0.5 * eyeSeperation, 0.0, 0.0), lookAt);
auto left_camera = vsg::Camera::create(left_relative_perspective, left_relative_view, viewport);

// Right eye camera - offset right, renders to cyan channels
auto right_relative_perspective = vsg::RelativeProjection::create(
    vsg::translate(shear, 0.0, 0.0), perspective);
auto right_relative_view = vsg::RelativeViewMatrix::create(
    vsg::translate(0.5 * eyeSeperation, 0.0, 0.0), lookAt);
auto right_camera = vsg::Camera::create(right_relative_perspective, right_relative_view, viewport);
```

Color Channel Masking

```
// Define view masks
vsg::Mask leftMask = 0x1;   // Red channel
vsg::Mask rightMask = 0x2;  // Green/Blue channels

// Create color blend states for each eye
auto left_colorBlendState = vsg::ColorBlendState::create();
left_colorBlendState->mask = leftMask;
left_colorBlendState->attachments[0].colorWriteMask = VK_COLOR_COMPONENT_R_BIT | VK_COLOR_COMPONENT_A_BIT;

auto right_colorBlendState = vsg::ColorBlendState::create();
right_colorBlendState->mask = rightMask;
right_colorBlendState->attachments[0].colorWriteMask = VK_COLOR_COMPONENT_G_BIT | VK_COLOR_COMPONENT_B_BIT | VK_COLOR_CC
```

Stereo Rendering Setup

```
auto renderGraph = vsg::RenderGraph::create(window);

// Render left view first (red channel)
auto left_view = vsg::View::create(left_camera, scene);
left_view->mask = leftMask;
renderGraph->addChild(left_view);

// Clear depth buffer between views
VkClearValue clearValue{};
clearValue.depthStencil = {0.0f, 0};
VkClearAttachment depth_attachment{VK_IMAGE_ASPECT_DEPTH_BIT, 1, clearValue};
VkClearRect rect{right_camera->getRenderArea(), 0, 1};
auto clearAttachments = vsg::ClearAttachments::create(
    vsg::ClearAttachments::Attachments{depth_attachment},
    vsg::ClearAttachments::Rects{rect}
);
renderGraph->addChild(clearAttachments);

// Render right view second (cyan channels)
auto right_view = vsg::View::create(right_camera, scene);
right_view->mask = rightMask;
renderGraph->addChild(right_view);
```

Dynamic Eye Separation

```
// Adjust eye separation based on viewing distance
double lookDistance = vsg::length(lookAt->center - lookAt->eye);
double horizontalSeperation = 0.5 * eyeSeperation;
horizontalSeperation *= (lookDistance / screenDistance);

// Update relative view matrices
left_relative_view->matrix = vsg::translate(horizontalSeperation, 0.0, 0.0);
right_relative_view->matrix = vsg::translate(-horizontalSeperation, 0.0, 0.0);
```

Pipeline State Modification for Stereo

Use the visitor pattern to modify existing graphics pipelines for stereo rendering:

```
class ReplaceColorBlendState : public vsg::Visitor
{
public:
    ReplaceColorBlendState(vsg::Mask leftMask, vsg::Mask rightMask) :
        leftMask(leftMask), rightMask(rightMask) {}

    void apply(vsg::BindGraphicsPipeline& bgp) override
    {
        auto gp = bgp.pipeline;

        // Find and replace ColorBlendState
        for (auto itr = gp->pipelineStates.begin(); itr != gp->pipelineStates.end(); ++itr)
        {
            if ((*itr)->is_compatible(typeid(vsg::ColorBlendState)))
            {
                auto colorBlendState = itr->cast<vsg::ColorBlendState>();
                gp->pipelineStates.erase(itr);

                // Add stereo-specific blend states
                auto left_colorBlendState = vsg::ColorBlendState::create(*colorBlendState);
                left_colorBlendState->mask = leftMask;
                left_colorBlendState->attachments[0].colorWriteMask = VK_COLOR_COMPONENT_R_BIT | VK_COLOR_COMPONENT_A_BI

                auto right_colorBlendState = vsg::ColorBlendState::create(*colorBlendState);
                right_colorBlendState->mask = rightMask;
                right_colorBlendState->attachments[0].colorWriteMask = VK_COLOR_COMPONENT_G_BIT | VK_COLOR_COMPONENT_B_E

                gp->pipelineStates.push_back(left_colorBlendState);
                gp->pipelineStates.push_back(right_colorBlendState);
                return;
            }
        }
    }

private:
    vsg::Mask leftMask, rightMask;
};

// Apply visitor to modify existing pipelines
ReplaceColorBlendState replaceVisitor(leftMask, rightMask);
scene->accept(replaceVisitor);
```

Key Stereo Rendering Concepts

- **RelativeProjection**: Applies additional transform to existing projection matrix
- **RelativeViewMatrix**: Applies additional transform to existing view matrix
- **View masks**: Control which objects render in which views
- **Color channel masking**: Restrict rendering to specific color channels
- **Depth buffer management**: Clear depth between stereo passes to prevent interference
- **Convergence**: Adjust projection shearing for comfortable stereo viewing

Animation

Content will be added as examples are documented

Input and UI

Event Handling

VSG provides a powerful event system that enables creation of custom camera controllers and interactive applications through the visitor pattern.

Custom Camera Controllers

Create specialized camera navigation by implementing custom event handlers:

```
class TurntableCamera : public vsg::Inherit<vsg::Visitor, TurntableCamera>
{
public:
    explicit TurntableCamera(vsg::ref_ptr<vsg::Camera> camera) :
        _camera(camera), _lookAt(camera->viewMatrix.cast<vsg::LookAt>()) {}

    // Core camera operations
    virtual void rotate(double angle, const vsg::dvec3& axis);
    virtual void zoom(double ratio);
    virtual void pan(const vsg::dvec2& delta);

    // Input event handling
    void apply(vsg::KeyPressEvent& keyPress) override
    {
        switch (keyPress.keyBase)
        {
            case vsg::KEY_w: pitchUp(); break;
            case vsg::KEY_s: pitchDown(); break;
            case vsg::KEY_a: turnLeft(); break;
            case vsg::KEY_d: turnRight(); break;
            case vsg::KEY_q: rollLeft(); break;
            case vsg::KEY_e: rollRight(); break;
        }
    }

    void apply(vsg::ButtonPressEvent& buttonPress) override
    {
        if (withinRenderArea(buttonPress))
        {
            _updateMode = ROTATE;
            _previousPointerEvent = vsg::ref_ptr{&buttonPress};
        }
    }

    void apply(vsg::MoveEvent& moveEvent) override
    {
        if (_updateMode == ROTATE && _previousPointerEvent)
        {
            auto delta = computeMovementDelta(moveEvent);

            // Convert screen space movement to rotation
            double angle = vsg::length(delta) * rotationSensitivity;
            vsg::dvec3 axis = vsg::normalize(vsg::dvec3(-delta.y, delta.x, 0.0));

            rotate(angle, axis);
        }

        _previousPointerEvent = vsg::ref_ptr{&moveEvent};
    }

    void apply(vsg::ScrollWheelEvent& scrollWheel) override
    {
        if (withinRenderArea(scrollWheel))
        {
            double zoomRatio = 1.0 + (scrollWheel.delta.y * zoomSensitivity);
            zoom(zoomRatio);
        }
    }

    void apply(vsg::TouchMoveEvent& touchMove) override
    {
        if (touchMove.touches.size() == 1)
        {
            // Single touch rotation
            handleSingleTouchRotation(touchMove.touches[0]);
        }
        else if (touchMove.touches.size() == 2)
        {
            // Two finger zoom/pan
            handleTwoFingerGestures(touchMove.touches);
        }
```

```
    }

    void apply(vsg::FrameEvent& frame) override
    {
        // Handle smooth camera animations
        updateCameraAnimation(frame.frameStamp->time);
    }

private:
    vsg::ref_ptr<vsg::Camera> _camera;
    vsg::ref_ptr<vsg::LookAt> _lookAt;

    enum UpdateMode { INACTIVE, ROTATE, PAN, ZOOM } _updateMode = INACTIVE;
    vsg::ref_ptr<vsg::PointerEvent> _previousPointerEvent;

    double rotationSensitivity = 0.01;
    double zoomSensitivity = 0.1;
};
```

Coordinate System Conversions

```
// Convert screen coordinates to normalized device coordinates (-1 to 1)
vsg::dvec2 screenToNDC(const vsg::PointerEvent& event)
{
    auto renderArea = _camera->getRenderArea();
    double x = (2.0 * (event.x - renderArea.offset.x)) / renderArea.extent.width - 1.0;
    double y = (2.0 * (event.y - renderArea.offset.y)) / renderArea.extent.height - 1.0;
    return vsg::dvec2(x, -y);  // Flip Y for typical graphics coordinate system
}

// Convert to turntable coordinates for camera manipulation
vsg::dvec3 ndcToTurntable(const vsg::dvec2& ndc)
{
    double theta = ndc.x * vsg::PI;        // Horizontal rotation
    double phi = ndc.y * vsg::PI * 0.5;    // Vertical rotation

    return vsg::dvec3(
        cos(phi) * cos(theta),
        cos(phi) * sin(theta),
        sin(phi)
    );
}
```

Multi-Window Input Handling

```
class MultiWindowCamera : public vsg::Inherit<vsg::Visitor, MultiWindowCamera>
{
public:
    // Map windows to coordinate offsets
    std::map<vsg::observer_ptr<vsg::Window>, vsg::ivec2> windowOffsets;

    void addWindow(vsg::ref_ptr<vsg::Window> window, const vsg::ivec2& offset = {})
    {
        windowOffsets[window] = offset;
    }

    std::pair<int32_t, int32_t> adjustedCoordinates(const vsg::PointerEvent& event) const
    {
        auto window_itr = windowOffsets.find(event.window);
        if (window_itr != windowOffsets.end())
        {
            auto& offset = window_itr->second;
            return {event.x + offset.x, event.y + offset.y};
        }
        return {event.x, event.y};
    }

    bool withinRenderArea(const vsg::PointerEvent& event) const
    {
        auto [x, y] = adjustedCoordinates(event);
```

```
        auto renderArea = _camera->getRenderArea();

        return (x >= renderArea.offset.x &&
                y >= renderArea.offset.y &&
                x < (renderArea.offset.x + renderArea.extent.width) &&
                y < (renderArea.offset.y + renderArea.extent.height));
    }
};
```

Smooth Camera Animation

```
class AnimatedViewpoint
{
public:
    void setViewpoint(vsg::ref_ptr<vsg::LookAt> target, double duration)
    {
        if (duration <= 0.0)
        {
            *_lookAt = *target;
            return;
        }

        _startTime = vsg::clock::now();
        _startLookAt = vsg::LookAt::create(*_lookAt);
        _endLookAt = target;
        _duration = duration;
        _animating = true;
    }

    void updateAnimation(vsg::time_point currentTime)
    {
        if (!_animating) return;

        auto elapsed = std::chrono::duration<double>(currentTime - _startTime).count();

        if (elapsed >= _duration)
        {
            *_lookAt = *_endLookAt;
            _animating = false;
        }
        else
        {
            double t = elapsed / _duration;
            t = smoothstep(t);  // Apply easing curve

            // Spherical interpolation for smooth rotation
            _lookAt->eye = vsg::mix(_startLookAt->eye, _endLookAt->eye, t);
            _lookAt->center = vsg::mix(_startLookAt->center, _endLookAt->center, t);
            _lookAt->up = vsg::normalize(vsg::mix(_startLookAt->up, _endLookAt->up, t));
        }
    }

private:
    vsg::ref_ptr<vsg::LookAt> _lookAt;
    vsg::time_point _startTime;
    vsg::ref_ptr<vsg::LookAt> _startLookAt, _endLookAt;
    double _duration = 0.0;
    bool _animating = false;

    // Hermite interpolation for smooth easing
    static double smoothstep(double t)
    {
        return t * t * (3.0 - 2.0 * t);
    }
};
```

Gimbal Lock Avoidance

```
void applyRotationWithGimbalAvoidance(const vsg::dvec3& rotation,
                                      const vsg::dvec3& globalUp = {0, 0, 1})
```

```
{
    auto lookVector = vsg::normalize(_lookAt->center - _lookAt->eye);

    // Compute rotation axes to avoid gimbal lock
    vsg::dvec3 horizontalAxis = vsg::normalize(vsg::cross(lookVector, globalUp));
    vsg::dvec3 verticalAxis = vsg::normalize(vsg::cross(horizontalAxis, lookVector));

    // Apply rotations around computed axes
    auto horizontalRotation = vsg::rotate(rotation.x, horizontalAxis);
    auto verticalRotation = vsg::rotate(rotation.y, verticalAxis);

    // Combine rotations and update camera
    auto combinedRotation = horizontalRotation * verticalRotation;
    double distance = vsg::length(_lookAt->center - _lookAt->eye);

    auto newDirection = combinedRotation * lookVector;
    _lookAt->eye = _lookAt->center - newDirection * distance;
}
```

Integration with Viewer

```
// Create and configure custom camera controller
auto viewer = vsg::Viewer::create();
auto camera = vsg::Camera::create(perspective, lookAt, viewport);

auto customController = TurntableCamera::create(camera);
customController->addWindow(window);

// Configure input sensitivity
customController->rotationSensitivity = 0.005;
customController->zoomSensitivity = 0.1;

// Add preset viewpoints
customController->addKeyViewpoint(vsg::KEY_1, frontView, 1.0);
customController->addKeyViewpoint(vsg::KEY_2, sideView, 1.0);
customController->addKeyViewpoint(vsg::KEY_3, topView, 1.0);

viewer->addEventHandler(customController);
```

Touch Gesture Recognition

```
void handleTouchGestures(const std::vector<vsg::TouchEvent>& touches)
{
    if (touches.size() == 2)
    {
        // Two finger gestures
        auto& touch1 = touches[0];
        auto& touch2 = touches[1];

        // Current gesture state
        vsg::dvec2 center = (vsg::dvec2(touch1.x, touch1.y) + vsg::dvec2(touch2.x, touch2.y)) * 0.5;
        double distance = vsg::length(vsg::dvec2(touch2.x - touch1.x, touch2.y - touch1.y));

        if (_previousTouchState.valid)
        {
            // Pan gesture - center movement
            auto panDelta = center - _previousTouchState.center;
            pan(panDelta * panSensitivity);

            // Zoom gesture - distance change
            if (_previousTouchState.distance > 0.0)
            {
                double zoomRatio = distance / _previousTouchState.distance;
                zoom(zoomRatio);
            }
        }

        // Store current state
        _previousTouchState.center = center;
        _previousTouchState.distance = distance;
```

```
        _previousTouchState.valid = true;
    }
    else
    {
        _previousTouchState.valid = false;
    }
}
```

ImGui Integration

Content will be added as examples are documented

Threading

Multi-threaded Rendering

Content will be added as examples are documented

Dynamic Loading

Content will be added as examples are documented

Device Management

VSG typically handles Vulkan device creation automatically, but provides mechanisms for manual device selection and configuration when needed.

Physical Device Enumeration

```
// Create window to access Vulkan instance
auto window = vsg::Window::create(windowTraits);
auto instance = window->getOrCreateInstance();
auto surface = window->getOrCreateSurface();

// Get all available physical devices
auto physicalDevices = instance->getPhysicalDevices();

for (auto& physicalDevice : physicalDevices)
{
    auto properties = physicalDevice->getProperties();

    // Check device compatibility
    auto [graphicsFamily, presentFamily] = physicalDevice->getQueueFamily(
        windowTraits->queueFlags, surface);

    bool suitable = (graphicsFamily >= 0 && presentFamily >= 0);

    std::cout << "Device: " << properties.deviceName
              << ", Type: " << properties.deviceType
              << ", Suitable: " << suitable << std::endl;
}
```

Manual Device Selection

```
// Select physical device by index
auto physicalDevices = instance->getPhysicalDevices();
if (deviceIndex < physicalDevices.size())
{
    auto selectedDevice = physicalDevices[deviceIndex];
    window->setPhysicalDevice(selectedDevice);
}

// Prefer discrete GPU over integrated
auto physicalDevice = instance->getPhysicalDevice(
    windowTraits->queueFlags,
    surface,
    {VK_PHYSICAL_DEVICE_TYPE_DISCRETE_GPU, VK_PHYSICAL_DEVICE_TYPE_INTEGRATED_GPU}
);
window->setPhysicalDevice(physicalDevice);
```

Queue Family Analysis

```
auto& queueFamilyProperties = physicalDevice->getQueueFamilyProperties();

for (size_t i = 0; i < queueFamilyProperties.size(); ++i)
{
    auto& prop = queueFamilyProperties[i];

    // Check capabilities
    bool hasGraphics = prop.queueFlags & VK_QUEUE_GRAPHICS_BIT;
    bool hasCompute = prop.queueFlags & VK_QUEUE_COMPUTE_BIT;
    bool hasTransfer = prop.queueFlags & VK_QUEUE_TRANSFER_BIT;

    std::cout << "Queue Family " << i
              << ", Count: " << prop.queueCount
              << ", Graphics: " << hasGraphics
              << ", Compute: " << hasCompute << std::endl;
}
```

Custom Device Creation

```
// Specify required extensions
vsg::Names deviceExtensions;
deviceExtensions.push_back(VK_KHR_SWAPCHAIN_EXTENSION_NAME);

// Configure queue settings
vsg::QueueSettings queueSettings{
    vsg::QueueSetting{graphicsFamily, {1.0}},
    vsg::QueueSetting{presentFamily, {1.0}}
};

// Create logical device
auto device = vsg::Device::create(
    physicalDevice,
    queueSettings,
    validatedLayers,
    deviceExtensions,
    deviceFeatures,
    allocationCallbacks
);

window->setDevice(device);
```

Device Type Categories

- **VK_PHYSICAL_DEVICE_TYPE_DISCRETE_GPU** (2) - Dedicated graphics card
- **VK_PHYSICAL_DEVICE_TYPE_INTEGRATED_GPU** (1) - Integrated graphics
- **VK_PHYSICAL_DEVICE_TYPE_VIRTUAL_GPU** (3) - Virtual machine GPU
- **VK_PHYSICAL_DEVICE_TYPE_CPU** (4) - Software renderer
- **VK_PHYSICAL_DEVICE_TYPE_OTHER** (0) - Unknown type

Best Practices

- Let VSG handle device selection automatically unless specific requirements exist
- Prefer discrete GPUs for performance-critical applications
- Check queue family support for required operations
- Validate device extension availability before use
- Consider memory heaps and limits for resource-intensive applications

Advanced Topics

Ray Tracing

Content will be added as examples are documented

Compute Shaders

Content will be added as examples are documented

Volume Rendering

Content will be added as examples are documented

Mesh Shaders

Content will be added as examples are documented

This guide is continuously updated as examples are documented. Each section will be expanded with practical code examples and best practices.