Overview

This PR is intended for adding a SDF renderer for the game. The architecture/design and experiments info will be added here. The main goal is to separate the rendering code into a separate sdf_renderer module. We manage any kind of SDF rendering in this module and expose minimum functions (at more 3-4) to be called in the engine_main.c to kick off rendering.

This PR handles the abstraction of SDF render to draw SDF primitives and supports operations and blending between complex SDF primitives using a node based scene representation.

Design

Currently backend only supports OpenGL 4.1 (sorry no compute shaders yet) this will directly contain API code as well. This is a simple module that handles everything internally. We might write a vk_backend for the sake of compute shaders and to run the game on macOS/win/linux etc.

Terminology

This was clarified by @PsychedelicOrange.

Primitive: A primitive is the most basic SDF shapes, they form the building block of complex shapes in the game.

Object: A Object is made up of one or more primitives and objects, they enable hierarchical combinations of fifteen shapes and can be combined using different blend operations like union, intersection, smooth union, subtractions etc. We can use hierarchal binary combinations of primitives and objects to create very complex shapes.

Node: A node is the building block of the scene, it's a DS to encapsulate primitive/object into a single structure. Every renderable in the scene is represented as nodes. Each node contains the node type, it's data and sphere bounds and other internal data.

Scene: A scene is a collections of nodes that will be used to draw primitives/objects, it also contains references nodes that are skipped from drawing and are used in the ray marching shader to blend and draw and more complex shapes. Scene contains a linear array of nodes.

Blending: This tells us how two primitives/object will be combined using the following operations: union, subtractions, smooth union etc. This tells how to build an object in the game.

Operations: These can act on primitives and objects alike. (eg. transformation, distortion, elongation, rotation, etc. These can either be applied on the final evaluated shapes or inserted during the intermediate steps.

SDF Renderer

Goal: We need to be able to draw primitives as well as support operation between the nodes (such as union, intersection, difference etc.). The most efficient way would be to do this in a compute shader and write it to a 3D texture before the rendering starts and use a hybrid approach to draw the dynamic nodes using a Pixel Shader approach.

Solution: The idea is to combine primitives and operation as scene nodes and use primitive refs to chain operations to draw more complex SDFs in a single draw call.

We start off with a simple Pixel Shader approach and build on it later. We begin with defining a enum `SDF_PrimitiveType` that the renderer can use to draw the SDF primitives, these are the building blocks of the SDF renderer.

To handle more complex shapes (for ex. SDF editor)., we can add more primitives and game specific shapes in the enum and define the SDF functions for the same in the ray marching shader. We can also extend this to take in custom functions but that is beyond the scope of this PR.

```
typedef enum SDF_PrimitiveType
{
    SDF_PRIM_Cube = 0,
    SDF_PRIM_Sphere,
    SDF_PRIM_Capsule,
    SDF_PRIM_Cylinder,
    // Game Specific Primitives
    SDF_PRIM_Planet,
    SDF_PRIM_Spaceship,
    SDF_PRIM_Bullet,
    SDF_PRIM_Ghost
} SDF_PrimitiveType;
```

Next we define the possible operations and blending between the SDF functions, these are used to create new shapes and combine SDFs.

```
typedef enum SDF_BlendType {
    SDF_BLEND_UNION,
    SDF_BLEND_SMOOTH_UNION,
    SDF_BLEND_INTERSECTION,
    SDF_BLEND_SUBTRACTION
} SDF_BlendType;

typedef enum SDF_Operation {
    SDF_OP_TRANSLATE,
    SDF_OP_ROTATE,
    SDF_OP_SCALE,
    SDF_OP_DISTORTION,
    SDF_OP_ELONGATION
} SDF_Operation;
```

Now we can either draw the primitive as is or use a **linear combination of primitives** to draw more complex shapes. So we define 2 structs one to handle the primitives and other to handle complex shapes and call it object as mentioned earlier in the terminology section.

```
typedef struct SDF_Primitive {
    SDF_PrimitiveType type;
    Transform transform; // Position, Rotation, Scale
    SDF_Material material;
    // add more props here combine them all or use a new struct/union to simplify primitive attributes
    vec4 params;
} SDF_Primitive;

typedef struct SDF_Object {
    SDF_BlendType type;
    int prim_a;                // Index of the left child in the SDF node pool
    int prim_b;               // Index of the right child in the SDF node pool
} SDF_Object;
```

These are the renderer structs that help the SDF renderer to draw SDFs.

SDF Scene

Now we need a way to store the SDF primitives and operation efficiently, the `SDF_Scene` stores a hierarchial node array of primitives and operations for the shader to handle.

```
typedef enum SDF_NodeType {
    SDF_NODE_PRIMITIVE,
    SDF_NODE_OBJECT
} SDF_NodeType;
```

Each node can either just draw a primitive or a complex shape using operations, this shape in turn can be linearly tagged along to create new shapes, so this design facilitates a chain-rule like mechanism to support SDF combinations without having to manage everything from within the shader. **For primitives that will be combined using operation, we provide separate API functions to add them as ref nodes to the scene instead of primitives that can be drawn individually.**

```
typedef struct SDF_Node {
    SDF_NodeType type;
    union {
        SDF_Primitive primitive;
        SDF_Operation operation;
    };
} SDF_Node;
```

`SDF_Node` is where all the magic happens for making more complex combinations. See the union, so we either ask the renderer_sdf to draw just a primitive as-is or evaluate the operations and combine the results to draw a more complex shape. For example to draw a 3-sphere metaball you can do something like this during the scene definition.

```
// Meta ball of 3 spheres
  {
      SDF_Primitive sphere1 = {
          .type      = SDF_PRIM_Sphere,
          .transform = {
              .position = {{demoStartX, 0.0f, 0.0f}},
              .rotation = {0.0f, 0.0f, 0.0f, 0.0f},
              .scale    = {{0.25f, 0.0f, 0.0f}}},
          .material = {.diffuse = {0.5f, 0.7f, 0.3f, 1.0f}}};
      int prim1 = sdf_scene_add_primitive(scene, sphere1);
      (void) prim1;

      sphere1.transform.position = (vec3s){{demoStartX + 0.5f, 0.5f, 0.0f}};
      int prim2                  = sdf_scene_add_primitive(scene, sphere1);
      (void) prim2;

      sphere1.transform.position = (vec3s){{demoStartX + 1.0f, 0.0f, 0.0f}};
      int prim3                  = sdf_scene_add_primitive(scene, sphere1);
      (void) prim3;

      sphere1.transform.position = (vec3s){{demoStartX + 0.5f, -0.5f, 0.0f}};
      int prim4                  = sdf_scene_add_primitive(scene, sphere1);
      (void) prim4;

      SDF_Object meta_def = {
          .type   = SDF_BLEND_SMOOTH_UNION,
          .prim_a = sdf_scene_add_object(scene, (SDF_Object){.type = SDF_BLEND_SMOOTH_UNION, .prim_a = prim1, .prim_b = prim2}),
          .prim_b = sdf_scene_add_object(scene, (SDF_Object){.type = SDF_BLEND_SMOOTH_UNION, .prim_a = prim3, .prim_b = prim4})};

      sdf_scene_add_object(scene, meta_def);
      demoStartX += 2.0f;
  }
```

This API is smart enough to mark some nodes `ref_nodes` and skip draw calls. whenever we use the function `sdf_scene_add_object` this functions will mark it's child nodes as `ref_node` , this way we can skip drawing the nodes that re references by complex objects and draw dangling primitives, the complex object will be handles in the ray marching shaders using a GPU stack simulation to create the final SDF evaluation.

```
typedef struct SDF_Scene {
    SDF_Node nodes[MAX_SDF_NODES];
    int node_count; // Number of active nodes after culling
} SDF_Scene;
```

The `SDF_Scene` struct contains all the nodes and references b/w them and primitives that will be passed to render for final drawing. This is uploaded as is to the GPU. We use a bindless mechanism to index into the node that we will be drawing, since we can skip some nodes, we can track the current index using `curr_node_idx` as a uniform/push_constant and pass it to the shader to push nodes onto the GPU simulated stack.

RayMarching Shader

Primitives like spheres and boxes are defined in the shader, and operations like union and intersection combine their SDFs. The SDF renderer parses the scene structure, performs node culling on the CPU, uploads active nodes to GPU , and executes the raymaching shader that evaluates the scene. This is how it is handles in the ray marching shader. To evaluate and combine hierarchical SDFs we need recursion, but due to the absence of stack and functions calls on GPU we need to simulate a stack on our own. More info on this will be uploaded in the docs section. But here's the base implementation of the shader.

More detailed explanation: https://github.com/Pikachuxxxx/Notes/blob/master/GPUSDFs/CombiningSDFsOnGPUAnimsSplit.pdf

Note: careful of the order or primitives when using blend operations like subtraction/intersection, usually SDF_B- SDF_A. Change the order to get the correct results.

```
////////////////////////////////////////////////////////////////////////////////
// Iterative Scene SDF Evaluation
struct hit_info
{
    float d;
    SDF_Material material;
};

struct blend_node {
    int blend;
    int node;
};

hit_info sceneSDF(vec3 p) {
    hit_info hit;
    hit.d = RAY_MAX_STEP;

    // Explicit stack to emulate tree traversal
    blend_node stack[MAX_STACK_SIZE];
    int sp = 0; // Stack pointer
    stack[sp++] = blend_node(SDF_BLEND_UNION, curr_draw_node_idx);

    while (sp > 0) {
        blend_node curr_blend_node = stack[--sp]; // Pop node index
        if (curr_blend_node.node < 0) continue;

        SDF_Node node = nodes[curr_blend_node.node];

        float d = RAY_MAX_STEP;
        // Evaluate the current node
        if(node.nodeType == 0) {
            if (node.primType == 0) { // Sphere
                d = sphereSDF(p, node.pos.xyz, node.scale.x);
            } else if (node.primType == 1) { // Box
                d = boxSDF(p, node.pos.xyz, node.scale.xyz);
            }

            // Apply the blend b/w primitives
            if (curr_blend_node.blend == SDF_BLEND_UNION) {
                hit.d = unionOp(hit.d, d);
            } else if (curr_blend_node.blend == SDF_BLEND_SMOOTH_UNION) {
                hit.d = opSmoothUnion(hit.d, d, 0.5f);
            } else if (curr_blend_node.blend == SDF_BLEND_INTERSECTION) {
                hit.d = intersectOp(hit.d, d);
            } else if (curr_blend_node.blend == SDF_BLEND_SUBTRACTION) {
                hit.d = subtractOp(hit.d, d);
            }
        }
        // it it's an Object process to blend and do stuff etc.
        else {
            // Push child nodes onto the stack
            if (sp < MAX_STACK_SIZE - 2) {
                if (node.prim_b >= 0) stack[sp++] = blend_node(node.blend, node.prim_b);
                if (node.prim_a >= 0) stack[sp++] = blend_node(node.blend, node.prim_a);
            }
        }

        hit.material = node.material;
    }

    return hit;
}
////////////////////////////////////////////////////////////////////////////////
```

The SDF_node DS is re-used in the shader to construct a more complex SDFScene that can be ray marched to draw primitives. Since operations node is a complex of 2 or more operation this supports combining 2 or more SDFs in a single draw call, however, careful combining more can lead to quick perf bottlenecks.

References

- https://iquilezles.org/articles/distfunctions/
- https://jamie-wong.com/2016/07/15/ray-marching-signed-distance-functions/#the-raymarching-algorithm