Vortex2D Documentation

Release 1.0

Maximilian Maldacker

May 14, 2019

Contents

CHAPTER 1

2D real-time fluid engine

1.1 Introduction

This is a 2D engine with the goal of being used in real-time scenarios, e.g. video games, in the same manner as a Box2D.

It is a hybrid engine that uses particles for the advection part (moving the particles given their velocities) and a grid to resolve the incompressible constrains. However, the particles are not visible to the user and you only work with the grid directly.

The engine runs directly on the GPU using Vulkan compute shaders. The rendering is then also done with Vulkan. The grids are represented by textures and operations by the user are all done by drawing shapes on the textures. The engine itself is written in C++ and it provides a simple wrapper around Vulkan and the basic rendering functionalities: shapes, textures, sprites, blending, render textures, etc.

1.2 Table of content

1.2.1 Setup

Vortex2D is multi-platform and currently supports the following:

- Windows
- Linux
- macOS
- iOS

CMake is used to generate the appropriate build scripts for each platform. The dependencies, which are fetched when calling cmake, are **glm** and **SPIRV-cross**. The tests use **gtest** and the examples use **glfw**.

The only dependency required is python. There a several variables that can be used to configure:

CMake	Builds
VORTEX2D_ENABLE_TESTS	builds the tests
VORTEX2D_ENABLE_EXAMPLES	builds the examples
VORTEX2D_ENABLE_DOCS	builds the documentation

The main library is built as a dll on windows, shared library on linux and (dynamic) framework on macOS/iOS.

Prerequisite

Following dependencies are necessary:

- Python
- glslangValidator (comes with Vulkan SDK)

Following minimum compilers are necessary:

- GCC 5.4 or later
- MSVC 2015 or later
- Clang 3.4 or later

Windows

To build on windows, *cmake-gui* is the easiest to use. Only the variables specified above should be changed.

Linux

The package *xorg-dev* might need to first be installed. Again, regular cmake commands should be use to configure cmake:

```
cmake ..
```

macOS

In addition to the normal variables, we need to specify the location of MoltenVK and the glslang compiler. The glslang compiler can be downloaded from its project on github: https://github.com/KhronosGroup/glslang/releases

```
cmake .. -DMOLTENVK_DIR=path_to/MoltenVK/Package/Latest/MoltenVK/ -DGLSL_
↪VALIDATOR=path_to/bin/glslangValidator
```

iOS

The framework needs to signed on iOS, so the following variables need to be defined:

Variable	Value
CODE_SIGN_IDENTITY	"iPhone Developer"
DEVELOPMENT_TEAM_ID	set to the team id, can be found on the apple developer portal

In addition, the MoltenVK location has to be specified, and the toolchain:

```
cmake .. -DCMAKE_TOOLCHAIN_FILE=../cmake/ios.toolchain.cmake -DIOS_PLATFORM=OS -DIOS_
↪ARCH=arm64 -DENABLE_VISIBILITY=true -DGLSL_VALIDATOR=path_to/bin/glslangValidator -
↪DMOLTENVK_DIR=path_to_sdk/MoltenVK/ -DCODE_SIGN_IDENTITY="iPhone Developer" -
↪DDEVELOPMENT_TEAM_ID=XXXXXX
```

Documentation

To build the documentation the following is required:

- doxygen
- sphinx
- sphinx_rtd_theme
- sphinx breathe

1.2.2 Rendering

Initialization

The rendering API is very basic and supports only the most basic functionality.

Create an instance of *Vortex2D::Renderer::Instance* which is then used to create an instance of *Vortex2D::Renderer::Device*.

The device is then used to create any other object. The main one is the *Vortex2D::Renderer::RenderWindow* which is a window where to render sprites and polygons. The function `Vortex2D::Fluid::RenderWindow::Display()` is then used to present the result to the screen.

```
Vortex2D::Renderer::Instance instance("Application name", extensions); // pass list␣
↪of required extensions
Vortex2D::Renderer::Device device(instance.GetPhysicalDevice(), surface);

Vortex2D::Renderer::RenderWindow window(device, surface, width, height);
```

Note that the instance requires a list of extensions necessary to create a window. With GLFW they can be retrived as:

```
std::vector<const char*> GetGLFWExtensions()
{
    std::vector<const char*> extensions;
    unsigned int glfwExtensionCount = 0;
    const char** glfwExtensions;

    // get the required extensions from GLFW
    glfwExtensions = glfwGetRequiredInstanceExtensions(&glfwExtensionCount);
    for (unsigned int i = 0; i < glfwExtensionCount; i++)
    {
        extensions.push_back(glfwExtensions[i]);
    }

    return extensions;
}
```

In addition, you also need to create a surface which can be also done with the help of GLFW:

```
vk::UniqueSurfaceKHR GetGLFWSurface(GLFWwindow* window, vk::Instance instance)
{
    // create surface
    VkSurfaceKHR surface;
    if (glfwCreateWindowSurface(static_cast<VkInstance>(instance), window, nullptr, &
↪surface) != VK_SUCCESS)
    {
        throw std::runtime_error("failed to create window surface!");
    }

    return vk::UniqueSurfaceKHR(surface, vk::SurfaceKHRDeleter{instance});
}
```

Render Targets

To be able to render, we need to record *Vortex2D::Renderer::RenderCommand* on a *Vortex2D::Renderer::RenderTarget*. There are two implementations of it:

- *Vortex2D::Renderer::RenderWindow*
- *Vortex2D::Renderer::RenderTexture*

You can render implementations of the abstract class *Vortex2D::Renderer::Drawable*, which get recorder in the render command. To actually render it on the render target, the submit function needs to be called. Note, it can be called repeatedly (e.g. over several frames).

In addition, the blend state needs to be passed in, see *Vortex2D::Renderer::ColorBlendState*.

Shapes

We are now ready to draw things on the screen. Let's start with some shapes like rectangles and circles:

```
Vortex2D::Renderer::Rectangle rectangle(device, {100.0f, 100.0f});
Vortex2D::Renderer::Ellipse circle(device, {50.0f, 50.0f});

auto blendMode = vk::PipelineColorBlendAttachmentState()
    .setBlendEnable(true)
    .setAlphaBlendOp(vk::BlendOp::eAdd)
    .setColorBlendOp(vk::BlendOp::eAdd)
    .setSrcColorBlendFactor(vk::BlendFactor::eSrcAlpha)
    .setSrcAlphaBlendFactor(vk::BlendFactor::eOne)
    .setDstColorBlendFactor(vk::BlendFactor::eOneMinusSrcAlpha)
    .setDstAlphaBlendFactor(vk::BlendFactor::eZero);

// note that rectangle, circle and render need to be alive for the duration of the␣
↪rendering
auto render = renderTarget.Record({rectangle, circle}, blendMode);
render.Submit();
```

Textures

Of course we can also render textures, using sprites.

```
Vortex2D::Renderer::Texture texture(device, 100, 100, vk::Format::eR8G8B8A8Unorm);
Vortex2D::Renderer::Sprite sprite(device, texture);
```

Transformations

The shapes and textures can be positioned, i.e. are transformable. You can set the following properties on them:

- Position
- Scale
- Rotation
- Anchor

As an example:

```
Vortex2D::Renderer::Ellipse circle(device, {50.0f, 50.0f});
circle.Colour = {0.0f, 0.0f, 1.0f, 1.0f};
circle.Position = {500.0f, 400.0f};
```

1.2.3 Level sets

A level set is a signed distance field. It's a field containing positive or negative value, where the values are 0 represent a contour, or border. This is used to represent shapes, the numbers give you the distance to the shape border. It's the fundamental way that we represent the area of a fluid and the area of the obstacles, i.e. the boundaries.

The level set is represented simply as a float texture. To set the level set, we simply render on that texture. This means that the class *Vortex2D::Fluid::LevelSet* inherits *Vortex2D::Renderer::RenderTexture*.

Basic shapes

There is a list of basic shapes that can be used to render on a level set:

- *Vortex2D::Fluid::Rectangle*
- *Vortex2D::Fluid::Polygon*
- *Vortex2D::Fluid::Circle*

They are used the same way as regular drawable shapes, i.e.

```
Vortex2D::Fluid::Rectangle rectangle(device, {100.0f, 100.0f});
rectangle.Position = {40.0f, 60.0f};

Vortex2D::Fluid::LevelSet levelSet(device, {400, 400});
auto renderCmd = levelSet.Record({rectangle});
renderCmd.Submit(); // note that renderCmd and rectangle have to be alive untill the␣
↪rendering is done
```

Combining shapes

Multiple shapes can be combined together to build the level set. You can either take the union or the intersection when rendering. This happens by using certain blend states which are:

- `Vortex2D::Renderer::IntersectionBlend`
- `Vortex2D::Renderer::UnionBlend`

After combining several shapes, the resulting float texture is not a signed distance field. It needs to be reinitialised which is simply done by calling *Vortex2D::Fluid::LevelSet::Reinitialise()*.

1.2.4 World

The world classes are the centre of the engine, where the fluid gets animated. They contain essentially three fields:

- The velocity field
- The liquid phi field
- The solid phi field

The first one contain the velocity of the fluid at every point, the second one defines where the fluid is. This is a signed distance field where a negative value indicates this is a fluid location. Finally the last one contains the location of solid obstacles, again as a signed distance field where the negative values indicate the solid's location.

Each can be visualised as a texture with the getters:

```
Renderer::RenderTexture& GetVelocity();
DistanceField LiquidDistanceField();
DistanceField SolidDistanceField();
```

Of course, to get interesting fluid simulations, we need to set values on them. Setting the signed distance fields is straightword (see *Level sets*):

```
Renderer::RenderCommand RecordLiquidPhi(Renderer::RenderTarget::DrawableList␣
↪drawables);
Renderer::RenderCommand RecordStaticSolidPhi(Renderer::RenderTarget::DrawableList␣
↪drawables);
```

Note that this only has to be done once.

For velocities however, the simulation needs to set the velocities at a specific time during the simulation, so instead of ourselves calling *Vortex2D::Renderer::RenderCommand::Submit()* we pass the *Vortex2D::Renderer::RenderCommand()* to the `World::Fluid::World()` class:

```
Renderer::RenderCommand RecordVelocity(Renderer::RenderTarget::DrawableList␣
↪drawables);
void SubmitVelocity(Renderer::RenderCommand& renderCommand);
```

Stepping through the simulation is done with the *Vortex2D::Fluid::World::Step()* function, which takes as parameter the number of iterations used in the linear solver. This can either be a fixed number of steps, or until the error reaches a certain threshhold.

```
auto iterations = Fluid::FixedParams(12);
world.Step(iterations);
```

Smoke World

This is a type of fluid simulation where the fluid area doesn't move. This is used to simulate smoke type effects by having a colored texture be advected by the velocity field.

The class *Vortex2D::Fluid::Density* is used for this, it is simply a texture that can be rendered (i.e. a sprite).

The simulation is setup as so:

```
Fluid::Density density(device, size, vk::Format::eR8G8B8A8);
Fluid::SmokeWorld world(device, size, 0.033);
world.FieldBind(density);
```

Water World

This is a classical water type of fluid simulation. This has a fluid area which evoles over time, i.e. a area of water moving. The area of water and non-water can be specified by rendering onto the word, where each pixel indicates the number of particles to add/substract.

```
Renderer::RenderCommand RecordParticleCount(Renderer::RenderTarget::DrawableList␣
↪drawables);
```

The constraint is that the drawable needs to render integer values, which is provided for example by `Vortec2D::Renderer::IntRectangle` and used:

```
Renderer::IntRectangle fluid(device, {150.0f, 50.0f});
fluid.Position = {50.0f, 25.0f};
fluid.Colour = glm::vec4(4); // can also be -4

world.RecordParticleCount({fluid}).Submit().Wait();
```

1.2.5 Rigid body

Rigid bodies are the way to have dynamic interations with the fluid (other then changing the velocity field directly). Vortex2D only provides a way to get current forces applied to the rigidbodies, and applying velocities from the rigidbody to the fluid. Updating the rigidbody's forces, velocities and position needs to be done by a seperate engine, such as Box2D.

Rigidbodies have three types:

- Static
- Weak
- Strong

Static bodies

Static bodies act on the fluid, but the fluid doesn't act on the fluid. They have a velocity that is imparted on the fluid. Think of motorized objects pushing through the fluid.

Weak/Strong bodies

Weak rigidbodies are affected by the fluid. They can also in turn, affect the fluid, which is called a strong coupling with the fluid.

Rigid body updates

Mass and velocity is set using simple setter functions:

```
Rigidbody rigidbody(device, size, drawable, type);
rigidbody.SetMassData(mass, inertia);
rigidbody.SetVelocities(velocity, angle);
```

Position and orientation is updated the same as with shapes:

```
rigidbody.Position = {100.0f, 100.0f}
rigidbody.Rotation = 43.0f;
```

Rigid body coupling

To have the fluid influence the rigid bodies and vice versa, two functions need to be implemented by deriving:

- *Vortex2D::Fluid::RigidBody::ApplyForces()*
- *Vortex2D::Fluid::RigidBody::ApplyVelocities()*

The first one has forces from the fluid applied to the rigidbody. The second has velocities from the rigidbody applied to the fluid.

An example implementation with Box2D is as follow:

```
void Box2DRigidbody::ApplyForces()
{
  if (GetType() & Vortex2D::Fluid::RigidBody::Type::eWeak)
  {
    auto force = GetForces();
    b2Vec2 b2Force = {force.velocity.x, force.velocity.y};

    mBody->ApplyForceToCenter(b2Force, true);
    mBody->ApplyTorque(force.angular_velocity, true);
  }
}

void Box2DRigidbody::ApplyVelocities()
{
  auto pos = mBody->GetPosition();
  Position = {pos.x, pos.y};
  Rotation = glm::degrees(mBody->GetAngle());

  if (GetType() & Vortex2D::Fluid::RigidBody::Type::eStatic)
  {
    glm::vec2 vel = {mBody->GetLinearVelocity().x, mBody->GetLinearVelocity().y};
    float angularVelocity = mBody->GetAngularVelocity();
    SetVelocities(vel, angularVelocity);
  }
}
```

Note that any rigidbody physics can be used: chipmonk, bullet, etc.

Engine updates

Finally the rigidbody also needs to be updates, in lock-step with the fluid simulation.

Again, this is done by implementing `Vortex2D::Fluid::RigidBody::Step()`.

An example implementation with Box2D:

```
  void Box2DSolver::Step(float delta)
  {
    const int velocityStep = 8;
    const int positionStep = 3;
```

(continues on next page)

(continued from previous page)

```
    mWorld.Step(delta, velocityStep, positionStep);
  }

The delta is the same used to create the world object.
```

1.2.6 Renderer API reference

Classes

- *Vortex2D::Renderer::Clear*
- *Vortex2D::Renderer::Drawable*
- *Vortex2D::Renderer::Ellipse*
- *Vortex2D::Renderer::GenericBuffer*
- *Vortex2D::Renderer::IndirectBuffer*
- *Vortex2D::Renderer::Instance*
- *Vortex2D::Renderer::IntRectangle*
- *Vortex2D::Renderer::Rectangle*
- *Vortex2D::Renderer::RenderState*
- *Vortex2D::Renderer::RenderTarget*
- *Vortex2D::Renderer::RenderTexture*
- *Vortex2D::Renderer::RenderWindow*
- *Vortex2D::Renderer::Sprite*
- *Vortex2D::Renderer::Timer*
- *Vortex2D::Renderer::Transformable*
- *Vortex2D::Renderer::UniformBuffer*
- *Vortex2D::Renderer::VertexBuffer*
- *Vortex2D::Renderer::Work*

API Reference

namespace Renderer

Typedefs

using DescriptorTypeBindings = std::map<uint32_t, vk::DescriptorType>

typedef std::vector<glm::vec2> **Path**

Functions

template<template<typename> class **BufferType**, typename **T**>
void **CopyTo**(*BufferType*<*T*> &*buffer*, *T* &*t*)
Copy the content of a buffer in an object.

template<template<typename> class **BufferType**, typename **T**>
void **CopyTo**(*BufferType*<*T*> &*buffer*, std::vector<*T*> &*t*)
Copy the content of a buffer to a vector. Vector needs to have the correct size already.

template<template<typename> class **BufferType**, typename **T**>
void **CopyFrom**(*BufferType*<*T*> &*buffer*, **const** *T* &*t*)
Copy the content of an object to the buffer.

template<template<typename> class **BufferType**, typename **T**>
void **CopyFrom**(*BufferType*<*T*> &*buffer*, **const** std::vector<*T*> &*t*)
Copy the content of a vector to the buffer.

void **BufferBarrier**(vk::Buffer *buffer*, vk::CommandBuffer *commandBuffer*, vk::AccessFlags *oldAccess*, vk::AccessFlags *newAccess*)
Inserts a barrier for the given buffer, command buffer and access.

Parameters

- `buffer`: the vulkan buffer handle
- `commandBuffer`: the command buffer to inserts the barrier
- `oldAccess`: old access
- `newAccess`: new access

bool **operator==**(**const** *ShaderLayout* &*left*, **const** *ShaderLayout* &*right*)

bool **operator==**(**const** *PipelineLayout* &*left*, **const** *PipelineLayout* &*right*)

void **Bind**(**const** *Device* &*device*, *DescriptorSet* &*dstSet*, **const** *PipelineLayout* &*layout*, **const** std::vector<*BindingInput*> &*bindingInputs*)
Bind the resources (buffer or texture/sampler) to a *DescriptorSet*.

Parameters

- `device`: vulkan device
- `dstSet`: vulkan descriptor set
- `layout`: pipeline layout
- `bindingInputs`: list of resources (buffer or texture/sampler)

bool **HasLayer**(**const** char **extension*, **const** std::vector<vk::LayerProperties> &*availableExtensions*)

bool **HasExtension**(**const** char **extension*, **const** std::vector<vk::ExtensionProperties> &*availableExtensions*)

bool **operator==**(**const** *GraphicsPipeline* &*left*, **const** *GraphicsPipeline* &*right*)

bool **operator==**(**const** *SpecConstInfo* &*left*, **const** *SpecConstInfo* &*right*)

template<typename **Type**>

SpecConstInfo::Value<*Type*> **SpecConstValue**(uint32_t *id*, *Type* &&*value*)
Constructs a specialization constant value.

template<typename ...**Args**>
SpecConstInfo **SpecConst**(*Args*&&... *args*)
Constructs a *SpecConstInfo* with given values of specialisation constants.

bool **operator==**(**const** *RenderState* &*left*, **const** *RenderState* *right*)

vk::DeviceSize **GetBytesPerPixel**(vk::Format *format*)
Gets the number of bytes per pixel given the format.

Return bytes per pixel

Parameters

- `format`: of texture

void **TextureBarrier**(vk::Image *image*, vk::CommandBuffer *commandBuffer*, vk::ImageLayout *oldLayout*, vk::AccessFlags *srcMask*, vk::ImageLayout *newLayout*, vk::AccessFlags *dstMask*)
Inserts a barrier for the given texture, command buffer and access.

Parameters

- `image`: the vulkan image handle
- `commandBuffer`: the vulkan command buffer
- `oldLayout`: old layout
- `srcMask`: old access
- `newLayout`: new layout
- `dstMask`: new access

ComputeSize **MakeStencilComputeSize**(**const** glm::ivec2 &*size*, int *radius*)
Create a *ComputeSize* for a stencil type shader.

Return calculate *ComputeSize*

Parameters

- `size`: the domain size
- `radius`: the stencil size

ComputeSize **MakeCheckerboardComputeSize**(**const** glm::ivec2 &*size*)
Create a *ComputeSize* for a checkerboard type shader.

Return calculate *ComputeSize*

Parameters

- `size`: the domain size

class AbstractShape : **public** Vortex2D::*Renderer*::*Shape*
#include <*Shapes.h*> An polygonal shape where the fragment shader can be specified for customisation.

Subclassed by *Vortex2D::Renderer::IntRectangle*, *Vortex2D::Renderer::Rectangle*

Public Functions

void **Initialize**(**const** *RenderState* &*renderState*)
Initialize the drawable for a particular state. This might include creating the correct pipeline. If it was already initialized, it will do nothing.

Parameters
- `renderState`: the state to initialize with.

void **Update**(**const** glm::mat4 &*projection*, **const** glm::mat4 &*view*)
Update the MVP matrix of the drawable.

Parameters
- `projection`: the projection matrix
- `view`: the view matrix

void **Draw**(vk::CommandBuffer *commandBuffer*, **const** *RenderState* &*renderState*)
Draw for the given render state. This has to be initialized before.

Parameters
- `commandBuffer`: the command buffer to record into.
- `renderState`: the render state to use.

class AbstractSprite : **public** Vortex2D::*Renderer*::*Drawable*, **public** Vortex2D::*Renderer*::*Transformable*
#include <*Sprite.h*> a *Sprite*, i.e. a drawable that can render a texture. The fragment shader can be specified for customisation.

Subclassed by *Vortex2D::Fluid::DistanceField*, *Vortex2D::Renderer::Sprite*

Public Functions

void **Initialize**(**const** *RenderState* &*renderState*)
Initialize the drawable for a particular state. This might include creating the correct pipeline. If it was already initialized, it will do nothing.

Parameters
- `renderState`: the state to initialize with.

void **Update**(**const** glm::mat4 &*projection*, **const** glm::mat4 &*view*)
Update the MVP matrix of the drawable.

Parameters
- `projection`: the projection matrix
- `view`: the view matrix

void **Draw**(vk::CommandBuffer *commandBuffer*, **const** *RenderState* &*renderState*)
Draw for the given render state. This has to be initialized before.

Parameters
- `commandBuffer`: the command buffer to record into.
- `renderState`: the render state to use.

struct BindingInput
#include <*DescriptorSet.h*> The texture/sampler or buffer that can be binded to a shader.

template<typename **T**>
class Buffer : **public** Vortex2D::*Renderer*::*GenericBuffer*
#include <*Buffer.h*> a storage buffer type of buffer

class Clear : **public** Vortex2D::*Renderer*::*Drawable*

#include <*Shapes.h*> A drawable that simply clears the target.

Public Functions

void **Initialize**(**const** *RenderState* &*renderState*)

Initialize the drawable for a particular state. This might include creating the correct pipeline. If it was already initialized, it will do nothing.

Parameters

- `renderState`: the state to initialize with.

void **Update**(**const** glm::mat4 &*projection*, **const** glm::mat4 &*view*)

Update the MVP matrix of the drawable.

Parameters

- `projection`: the projection matrix
- `view`: the view matrix

void **Draw**(vk::CommandBuffer *commandBuffer*, **const** *RenderState* &*renderState*)

Draw for the given render state. This has to be initialized before.

Parameters

- `commandBuffer`: the command buffer to record into.
- `renderState`: the render state to use.

struct ColorBlendState

#include <*RenderState.h*> The blend state and blend constant.

class CommandBuffer

#include <*CommandBuffer.h*> Can record commands, then submit them (multiple times). A fence can used to wait on the completion of the commands.

Public Functions

CommandBuffer(**const** *Device* &*device*, bool *synchronise* = true)

Creates a command buffer which can be synchronized.

Parameters

- `device`: vulkan device
- `synchronise`: flag to determine if the command buffer can be waited on.

CommandBuffer &**Record**(CommandFn *commandFn*)

Record some commands. The commads are recorded in the lambda which is immediately executed.

Parameters

- `commandFn`: a functor, or simply a lambda, where commands are recorded.

CommandBuffer &**Record**(**const** *RenderTarget* &*renderTarget*, vk::Framebuffer *framebuffer*, CommandFn *commandFn*)

Record some commands inside a render pass. The commads are recorded in the lambda which is immediately executed.

Parameters

- `renderTarget`: the render target which contains the render pass to record into
- `framebuffer`: the frame buffer where the render pass will render.
- `commandFn`: a functor, or simply a lambda, where commands are recorded.

CommandBuffer &**Wait**()
Wait for the command submit to finish. Does nothing if the synchronise flag was false.

CommandBuffer &**Reset**()
Reset the command buffer so it can be recorded again.

CommandBuffer &**Submit**(**const** std::initializer_list<vk::Semaphore> &*waitSemaphores* = {}, **const** std::initializer_list<vk::Semaphore> &*signalSemaphores* = {})
submit the command buffer

operator bool() **const**
explicit conversion operator to bool, indicates if the command was properly recorded and can be sumitted.

struct ComputeSize
#include <Work.h> Used for a compute shader, and defines the group size, local size and domain size.

Public Functions

ComputeSize(**const** glm::ivec2 &*size*, **const** glm::ivec2 &*localSize* = *GetLocalSize2D*())
Creates a *ComputeSize* using a 2D domain size and the default 2D local size.

Parameters
- `size`: the domain size
- `localSize`: the local size of the shader

ComputeSize(int *size*, int *localSize* = *GetLocalSize1D*())
Creates a *ComputeSize* using a 1D domain size and the default 1D local size.

Parameters
- `size`: the domain size
- `localSize`: the local size of the shader

Public Static Functions

static glm::ivec2 **GetLocalSize2D**()
The default local size for 2D compute shaders.

Return a 2d vector

static int **GetLocalSize1D**()
The default local size for 1D compute shaders.

Return a integer value

static glm::ivec2 **GetWorkSize**(**const** glm::ivec2 &*size*, **const** glm::ivec2 &*localSize* = *GetLocalSize2D*())
Computes the 2D group size given a domain size.

Return the group size
Parameters
- `size`: the domain size of the shader
- `localSize`: the local size of the shader

static glm::ivec2 **GetWorkSize**(int *size*, int *localSize* = *GetLocalSize1D*())
Computes the 1D group size given a domain size.

Return the group size

Parameters
- `size`: the domain size of the shader
- `localSize`: the local size of the shader

static *ComputeSize* **Default2D**()
A default *ComputeSize* using the default 2D local size. The domain size is (1,1)

Return a default compute size

static *ComputeSize* **Default1D**()
A default *ComputeSize* using the default 1D local size. The domain size is (1,1)

Return a default compute size

struct DescriptorImage
#include <DescriptorSet.h> The texture or sampler that can be binded to a shader.

struct DescriptorSet
#include <DescriptorSet.h> The binding of an object for a shader.

class Device
#include <Device.h> Encapsulation around the vulkan device. Allows to create command buffers, layout, bindings, memory and shaders.

struct DispatchParams
#include <Work.h> Parameters for indirect compute: group size, local size, etc.

struct Drawable
#include <Drawable.h> Interface of a drawable object.

Subclassed by *Vortex2D::Fluid::Circle*, *Vortex2D::Fluid::Polygon*, *Vortex2D::Renderer::AbstractSprite*, *Vortex2D::Renderer::Clear*, *Vortex2D::Renderer::Shape*

Public Functions

virtual void **Initialize**(**const** *RenderState* &*renderState*) = 0
Initialize the drawable for a particular state. This might include creating the correct pipeline. If it was already initialized, it will do nothing.

Parameters
- `renderState`: the state to initialize with.

virtual void **Update**(**const** glm::mat4 &*projection*, **const** glm::mat4 &*view*) = 0
Update the MVP matrix of the drawable.

Parameters
- `projection`: the projection matrix
- `view`: the view matrix

virtual void **Draw**(vk::CommandBuffer *commandBuffer*, **const** *RenderState* &*renderState*) = 0
Draw for the given render state. This has to be initialized before.

Parameters
- `commandBuffer`: the command buffer to record into.
- `renderState`: the render state to use.

struct DynamicDispatcher
#include <Device.h> A vulkan dynamic dispatcher that checks if the function is not null.

class Ellipse : **public** Vortex2D::*Renderer*::*Shape*

#include <*Shapes.h*> A solid colour ellipse. Implements the *Drawable* interface and *Transformable* interface.

Public Functions

void **Initialize**(**const** *RenderState* &*renderState*)

Initialize the drawable for a particular state. This might include creating the correct pipeline. If it was already initialized, it will do nothing.

Parameters

- `renderState`: the state to initialize with.

void **Update**(**const** glm::mat4 &*projection*, **const** glm::mat4 &*view*)

Update the MVP matrix of the drawable.

Parameters

- `projection`: the projection matrix
- `view`: the view matrix

void **Draw**(vk::CommandBuffer *commandBuffer*, **const** *RenderState* &*renderState*)

Draw for the given render state. This has to be initialized before.

Parameters

- `commandBuffer`: the command buffer to record into.
- `renderState`: the render state to use.

class GenericBuffer

#include <*Buffer.h*> A vulkan buffer which can be on the host or the device.

Subclassed by *Vortex2D::Renderer::Buffer< float >*, *Vortex2D::Renderer::Buffer< glm::ivec2 >*, *Vortex2D::Renderer::Buffer< glm::vec2 >*, *Vortex2D::Renderer::Buffer< int >*, *Vortex2D::Renderer::Buffer< Vortex2D::Fluid::Particle >*, *Vortex2D::Renderer::Buffer< Vortex2D::Fluid::RigidBody::Velocity >*, *Vortex2D::Renderer::Buffer< Vortex2D::Renderer::DispatchParams >*, *Vortex2D::Renderer::IndirectBuffer< Vortex2D::Renderer::DispatchParams >*, *Vortex2D::Renderer::UniformBuffer< glm::mat4 >*, *Vortex2D::Renderer::UniformBuffer< glm::vec2 >*, *Vortex2D::Renderer::UniformBuffer< glm::vec4 >*, *Vortex2D::Renderer::UniformBuffer< Size >*, *Vortex2D::Renderer::UniformBuffer< Vortex2D::Fluid::RigidBody::Velocity >*, *Vortex2D::Renderer::VertexBuffer< glm::vec2 >*, *Vortex2D::Renderer::VertexBuffer< Vortex2D::Renderer::AbstractSprite::Vertex >*, *Vortex2D::Renderer::Buffer< T >*, *Vortex2D::Renderer::IndexBuffer< T >*, *Vortex2D::Renderer::IndirectBuffer< T >*, *Vortex2D::Renderer::UniformBuffer< T >*, *Vortex2D::Renderer::VertexBuffer< T >*

Public Functions

void **CopyFrom**(vk::CommandBuffer *commandBuffer*, *GenericBuffer* &*srcBuffer*)

Copy a buffer to this buffer.

Parameters

- `commandBuffer`: command buffer to run the copy on.
- `srcBuffer`: the source buffer.

void **CopyFrom**(vk::CommandBuffer *commandBuffer*, *Texture* &*srcTexture*)

Copy a texture to this buffer.

Parameters
- `commandBuffer`: command buffer to run the copy on.
- `srcTexture`: the source texture

vk::Buffer **`Handle`**`()` **`const`**
The vulkan handle.

vk::DeviceSize **`Size`**`()` **`const`**
The size in bytes of the buffer.

void **`Resize`**(vk::DeviceSize *size*)
Resize the buffer. Invalidates the buffer handle.

Parameters
- `size`: buffer size

void **`Barrier`**(vk::CommandBuffer *commandBuffer*, vk::AccessFlags *oldAccess*, vk::AccessFlags *newAccess*)
Inserts a barrier for this buffer.

Parameters
- `commandBuffer`: the command buffer to run the barrier
- `oldAccess`: old access
- `newAccess`: new access

void **`Clear`**(vk::CommandBuffer *commandBuffer*)
Clear the buffer with 0.

Parameters
- `commandBuffer`: the command buffer to clear on

void **`CopyFrom`**(uint32_t *offset*, **`const`** void **data*, uint32_t *size*)
copy from data to buffer

Parameters
- `offset`: in the buffer
- `data`: pointer
- `size`: of data

void **`CopyTo`**(uint32_t *offset*, void **data*, uint32_t *size*)
copy buffer to data

Parameters
- `offset`: in the buffer
- `data`: pointer
- `size`: of data

`class GraphicsPipeline`
#include <Pipeline.h> graphics pipeline which caches the pipeline per render states.

Public Functions

GraphicsPipeline &**`Shader`**(vk::ShaderModule *shader*, vk::ShaderStageFlagBits *shaderStage*)
Set the shader.

Return *this
Parameters
- `shader`: the loaded shader

- `shaderStage`: shader state (vertex, fragment or compute)

GraphicsPipeline &**VertexAttribute**(uint32_t *location*, uint32_t *binding*, vk::Format *format*, uint32_t *offset*)

Sets the vertex attributes.

Return *this

Parameters

- `location`: location in the shader
- `binding`: binding in the shader
- `format`: vertex format
- `offset`: offset in the vertex

GraphicsPipeline &**VertexBinding**(uint32_t *binding*, uint32_t *stride*, vk::VertexInputRate *inputRate* = vk::VertexInputRate::eVertex)

Sets the vertex binding.

Return *this

Parameters

- `binding`: binding in the shader
- `stride`: stride in bytes
- `inputRate`: inpute rate

template<typename **T**>
class IndexBuffer : **public** Vortex2D::*Renderer*::*GenericBuffer*

#include <*Buffer.h*> a index buffer type of buffer

template<typename **T**>
class IndirectBuffer : **public** Vortex2D::*Renderer*::*GenericBuffer*

#include <*Buffer.h*> an indirect buffer type of buffer, used for compute indirect dispatch

class Instance

#include <*Instance.h*> Vulkan instance, which extensions enabled.

class IntRectangle : **public** Vortex2D::*Renderer*::*AbstractShape*

#include <*Shapes.h*> A solid colour rectangle as *Rectangle*, however uses integer colors and is meant to be drawn to a framebuffer wiht integer colours.

class LayoutManager

#include <*DescriptorSet.h*> Caches and creates layouts and bindings.

Public Functions

void **CreateDescriptorPool**(int *size* = 512)

Create or re-create the descriptor pool, will render invalid existing descriptor sets.

Parameters

- `size`: size of the pool

DescriptorSet **MakeDescriptorSet**(**const** *PipelineLayout* &*layout*)

Create the descriptor set given the layout.

Return built descriptor set

Parameters

- `layout`: pipeline/shader layout

vk::DescriptorSetLayout **GetDescriptorSetLayout**(**const** *PipelineLayout* &*layout*)

Create, cache and return a descriptor layout given the pipeline layout.

Return cached descriptor set layout
Parameters
- `layout`: pipeline layout

vk::PipelineLayout **GetPipelineLayout**(**const** *PipelineLayout* &*layout*)
create, cache and return a vulkan pipeline layout given the layout

Return vulkan pipeline layout
Parameters
- `layout`: pipeline layout

class PipelineCache
#include <Pipeline.h> Create pipelines using vulkan's pipeline cache.

Public Functions

void **CreateCache**()
Create the pipeline cache.

vk::Pipeline **CreateGraphicsPipeline**(**const** *GraphicsPipeline* &*builder*, **const** *RenderState* &*renderState*)
Create a graphics pipeline.

Return
Parameters
- `builder`:
- `renderState`:

vk::Pipeline **CreateComputePipeline**(vk::ShaderModule *shader*, vk::PipelineLayout *layout*, *SpecConstInfo* *specConstInfo* = {})
Create a compute pipeline.

Parameters
- `shader`:
- `layout`:
- `specConstInfo`:

struct PipelineLayout
#include <DescriptorSet.h> Represents the layout of a pipeline: vertex + fragment or compute.

class Rectangle : **public** Vortex2D::*Renderer*::*AbstractShape*
#include <Shapes.h> A solid colour rectangle defined by two triangles. Implements the *Drawable* interface and *Transformable* interface.

class RenderCommand
#include <CommandBuffer.h> A special command buffer that has been recorded by a *RenderTarget*. It can be used to submit the rendering. The object has to stay alive untill rendering is complete.

Public Functions

RenderCommand &**Submit**(**const** glm::mat4 &*view* = glm::mat4(1.0f))
Submit the render command with a transform matrix.

Return *this
Parameters
- `view`: a transform matrix

void **Wait**()
Wait for the render command to complete.

operator bool() **const**
explicit conversion operator to bool, indicates if the command was properly recorded and can be sumitted.

class RenderpassBuilder
#include <RenderTarget.h> Factory for a vulkan render pass.

Public Functions

RenderpassBuilder &**Attachement**(vk::Format *format*)
Format of the render pass.

Return
Parameters
- `format`:

RenderpassBuilder &**AttachementLoadOp**(vk::AttachmentLoadOp *value*)
operation to perform when loading the framebuffer (clear, load, etc)

Return
Parameters
- `value`:

RenderpassBuilder &**AttachementStoreOp**(vk::AttachmentStoreOp *value*)
operation to perform when storing the framebuffer (clear, save, etc)

Return
Parameters
- `value`:

RenderpassBuilder &**AttachementInitialLayout**(vk::ImageLayout *layout*)
Layout of the image to be before render pass.

Return
Parameters
- `layout`:

RenderpassBuilder &**AttachementFinalLayout**(vk::ImageLayout *layout*)
Layout of the image to be after render pass.

Return
Parameters
- `layout`:

RenderpassBuilder &**Subpass**(vk::PipelineBindPoint *bindPoint*)
Define subpass of the render pass.

Return
Parameters
- `bindPoint`:

RenderpassBuilder &**SubpassColorAttachment**(vk::ImageLayout *layout*, uint32_t *attachment*)
Set the color attachment with index.

Return

Parameters
- `layout`:
- `attachment`: index of the attachment

RenderpassBuilder &**Dependency**(uint32_t *srcSubpass*, uint32_t *dstSubpass*)
Dependency of the subpasses.

Return
Parameters
- `srcSubpass`:
- `dstSubpass`:

vk::UniqueRenderPass **Create**(vk::Device *device*)
Create the render pass.

Return
Parameters
- `device`:

struct RenderState
#include <RenderState.h> the various state to render to a target: size, render pass and blend.

Public Functions

RenderState(**const** *RenderTarget* &*renderTarget*)
Initialize for a render target with default blend.

Parameters
- `renderTarget`:

RenderState(**const** *RenderTarget* &*renderTarget*, *ColorBlendState* *blendState*)
Initialize for a render target with a given blend.

Parameters
- `renderTarget`:
- `blendState`:

struct RenderTarget
#include <RenderTarget.h> A target that can be rendered to. This is implemented by the *RenderWindow* and the *RenderTexture*.

Subclassed by *Vortex2D::Renderer::RenderTexture*, *Vortex2D::Renderer::RenderWindow*

class RenderTexture : **public** Vortex2D::*Renderer*::*RenderTarget*, **public** Vortex2D::*Renderer*::*Texture*
#include <RenderTexture.h> A render target that renders into a texture.

Subclassed by *Vortex2D::Fluid::Density*, *Vortex2D::Fluid::LevelSet*, *Vortex2D::Fluid::ParticleCount*, *Vortex2D::Fluid::Velocity*

class RenderWindow : **public** Vortex2D::*Renderer*::*RenderTarget*
#include <RenderWindow.h> Render to a swapchain, i.e. to the window/surface.

Public Functions

RenderWindow(**const** *Device* &*device*, vk::SurfaceKHR *surface*, uint32_t *width*, uint32_t *height*)
Initialize with a given surface and size.

Parameters
- `device`: vulkan device
- `surface`: vulkan surface
- `width`:
- `height`:

void **Display**()
Submits all the render command and present the surface for display.

class SamplerBuilder
#include <Texture.h> Factory for a vullkan sampler.

Public Functions

SamplerBuilder &**AddressMode**(vk::SamplerAddressMode *mode*)
Mode of the sampler: repeat, clamp, etc.

Return *this
Parameters
- `mode`: vulkan mode

SamplerBuilder &**Filter**(vk::Filter *filter*)
Filter of the sampler: linear, nearest, etc.

Return *this
Parameters
- `filter`: vulkan filter

vk::UniqueSampler **Create**(vk::Device *device*)
Create the vulkan sampler.

Return unique sampler
Parameters
- `device`: vulkan device

struct ShaderLayout
#include <DescriptorSet.h> Represents the layout of a shader (vertex, fragment or compute)

class Shape : **public** Vortex2D::*Renderer*::*Drawable*, **public** Vortex2D::*Renderer*::*Transformable*
#include <Shapes.h> *Shape* interface which is drawable, transformable and has a color.

Subclassed by *Vortex2D::Renderer::AbstractShape*, *Vortex2D::Renderer::Ellipse*

struct SpecConstInfo
#include <Pipeline.h> Defines and holds value of the specification constants for shaders.

class SpirvBinary
#include <Device.h> A binary SPIRV shader, to be feed to vulkan.

class Sprite : **public** Vortex2D::*Renderer*::*AbstractSprite*
#include <Sprite.h> A sprite that renders a texture with a simple pass-through fragment shader.

Subclassed by *Vortex2D::Fluid::Density*

class Texture
#include <Texture.h> A texture, or in vulkan terms, an image.

Subclassed by *Vortex2D::Renderer::RenderTexture*

Public Functions

void **CopyFrom**(**const** void **data*)
Copies width*heigh*bytesPerPixel amount of data.

Parameters
- `data`: source data

void **CopyTo**(void **data*)
Copies width*heigh*bytesPerPixel amount of data.

Parameters
- `data`: destination data

void **CopyFrom**(vk::CommandBuffer *commandBuffer*, *Texture* &*srcImage*)
Copies source texture in this texture.

Parameters
- `commandBuffer`: vulkan command buffer
- `srcImage`: source image

class Timer
#include <Timer.h> Calculates the ellapsed time on the GPU.

Public Functions

void **Start**(vk::CommandBuffer *commandBuffer*)
Start the timer after the current last command buffer.

Parameters
- `commandBuffer`: command buffer to write timestamp

void **Stop**(vk::CommandBuffer *commandBuffer*)
Start the timer after the current last command buffer.

Parameters
- `commandBuffer`: command buffer to write timestamp

void **Start**()
Start the timer after the current last command buffer.

void **Stop**()
Stop the timer after the current last command buffer.

void **Wait**()
Wait for *Start* and *Stop* to finish before retrieving the results.

uint64_t **GetElapsedNs**()
Get the elapsed time between the Start and Stop calls. Blocking function which will download the timestamps from the GPU.

Return timestamp in nanoseconds.

struct Transformable
#include <Transformable.h> Class to represent the transformation of an object: position, scale, rotation and anchor.

Subclassed by *Vortex2D::Fluid::Circle*, *Vortex2D::Fluid::Polygon*, *Vortex2D::Fluid::RigidBody*, *Vortex2D::Renderer::AbstractSprite*, *Vortex2D::Renderer::Shape*

Public Functions

const glm::mat4 &**GetTransform**() **const**
Returns the transform matrix.

void **Update**()
Update the transormation matrix.

Public Members

glm::vec2 **Position**
absolute position

glm::vec2 **Scale**
scale for the x and y components

float **Rotation**
Rotation in radians.

glm::vec2 **Anchor**
An offset to the position (used for centering a shape)

template<typename **T**>
class UniformBuffer : **public** Vortex2D::*Renderer*::*GenericBuffer*
#include <*Buffer.h*> a uniform buffer type of buffer

template<typename **T**>
class VertexBuffer : **public** Vortex2D::*Renderer*::*GenericBuffer*
#include <*Buffer.h*> a vertex buffer type of buffer

class Work
#include <*Work.h*> Represents a compute shader. It simplifies the process of binding, setting push constants and recording.

Public Functions

Work(**const** *Device* &*device*, **const** *ComputeSize* &*computeSize*, **const** *SpirvBinary* &*spirv*, **const** *SpecConstInfo* &*additionalSpecConstInfo* = {})
Constructs an object using a SPIRV binary. It is not bound to any buffers or textures.

Parameters
- `device`: vulkan device
- `computeSize`: the compute size. Can be a default one with size (1,1) or one with an actual size.
- `spirv`: binary spirv
- `additionalSpecConstInfo`: additional specialization constants

Bound **Bind**(**const** std::vector<*BindingInput*> &*inputs*)
Bind the buffers and/or textures.

Return a bound object, ready to be recorded in a command buffer.
Parameters
- `inputs`: a list of buffers and/or textures

Bound **Bind**(*ComputeSize computeSize*, **const** std::vector<*BindingInput*> &*inputs*)
Bind the buffers and/or textures. This overrides the provided compute size in *Work*.

Return a bound object, ready to be recorded in a command buffer.
Parameters
- `computeSize`: the compute shader compute size.
- `inputs`: a list of buffers and/or textures

class Bound
#include <Work.h> Is a bound version of *Work*. This means a buffer or texture was bound and this can be recorded in a command buffer.

Public Functions

template<typename ...**Args**>
void **PushConstant**(vk::CommandBuffer *commandBuffer*, *Args*&&... *args*)
Adds a constant value, i.e. a push constant.

Parameters
- `commandBuffer`: the command buffer where the compute work will also be recorded.
- `args`: the data to push. A total of 128 bytes can be used.

void **Record**(vk::CommandBuffer *commandBuffer*)
Record the compute work in this command buffer. This will also set two additional push constants: the 2D domain size.

Parameters
- `commandBuffer`: the command buffer to record into.

void **RecordIndirect**(vk::CommandBuffer *commandBuffer*, *IndirectBuffer*<*DispatchParams*> &*dispatchParams*)
Record the compute work in this command buffer. Use the provided parameters to run the compute shader.

Parameters
- `commandBuffer`: the command buffer to record into.
- `dispatchParams`: the indirect buffer containing the parameters.

namespace Detail

Functions

void **InsertSpecConst**(*SpecConstInfo* &*specConstInfo*)

template<typename **Arg**, typename ...**Args**>
void **InsertSpecConst**(*SpecConstInfo* &*specConstInfo*, *Arg* &&*arg*, *Args*&&... *args*)

1.2.7 Fluid API reference

Classes

- *Vortex2D::Fluid::Advection*
- *Vortex2D::Fluid::Circle*
- *Vortex2D::Fluid::ConjugateGradient*
- *Vortex2D::Fluid::Density*

- *Vortex2D::Fluid::Depth*
- *Vortex2D::Fluid::Diagonal*
- *Vortex2D::Fluid::DistanceField*
- *Vortex2D::Fluid::Extrapolation*
- *Vortex2D::Fluid::GaussSeidel*
- *Vortex2D::Fluid::IncompletePoisson*
- *Vortex2D::Fluid::Jacobi*
- *Vortex2D::Fluid::LevelSet*
- *Vortex2D::Fluid::LinearSolver*
- *Vortex2D::Fluid::LocalGaussSeidel*
- *Vortex2D::Fluid::Multigrid*
- *Vortex2D::Fluid::ParticleCount*
- *Vortex2D::Fluid::Polygon*
- *Vortex2D::Fluid::Preconditioner*
- *Vortex2D::Fluid::Pressure*
- *Vortex2D::Fluid::Rectangle*
- *Vortex2D::Fluid::Reduce*
- *Vortex2D::Fluid::ReduceJ*
- *Vortex2D::Fluid::ReduceMax*
- *Vortex2D::Fluid::ReduceSum*
- *Vortex2D::Fluid::RigidBody*
- *Vortex2D::Fluid::SmokeWorld*
- *Vortex2D::Fluid::Transfer*
- *Vortex2D::Fluid::Velocity*
- *Vortex2D::Fluid::WaterWorld*
- *Vortex2D::Fluid::World*

API Reference

namespace Fluid

Enums

enum VelocityOp

Operator when applying velocity to velocity field: add or set.

Values:

Add

Set

Functions

LinearSolver::*Parameters* **FixedParams** (unsigned *iterations*)
Create a linear solver parameters object with fixed solver type.

Return parameters

Parameters

- `iterations`: number of iterations to do

LinearSolver::*Parameters* **IterativeParams** (float *errorTolerance*)
Create a linear solver parameters object, solver will continue until error tolerance is reached.

Return parameters

Parameters

- `errorTolerance`: tolerance to reach before exiting

Variables

Renderer::*ColorBlendState* **IntersectionBlend**

Renderer::*ColorBlendState* **UnionBlend**

Renderer::*Clear* **BoundariesClear**

class Advection
#include <Advection.h> Advects particles, velocity field or any field using a velocity field.

Public Functions

Advection (**const** *Renderer*::*Device* &*device*, **const** glm::ivec2 &*size*, float *dt*, *Velocity* &*velocity*)
Initialize advection kernels and related object.

Parameters

- `device`: vulkan device
- `size`: size of velocity field
- `dt`: delta time for integration
- `velocity`: velocity field

void **AdvectVelocity** ()
Self advect velocity.

void **AdvectBind** (*Density* &*density*)
Binds a density field to be advected.

Parameters

- `density`: density field

void **Advect** ()
Performs an advection of the density field. Asynchronous operation.

void **AdvectParticleBind**(*Renderer*::*GenericBuffer* &*particles*, *Renderer*::*Texture* &*levelSet*, *Renderer*::*IndirectBuffer*<*Renderer*::*DispatchParams*> &*dispatchParams*)

Binds praticles to be advected. Also use a level set to project out the particles if they enter it.

Parameters

- `particles`: particles to be advected
- `levelSet`: level set to project out particles
- `dispatchParams`: contains number of particles

void **AdvectParticles**()

Advect particles. Asynchrounous operation.

class Cfl

#include <Cfl.h> Calculates the CFL number of the velocity field. It's an indication on how to choose your time step size. Ideally, the time step should be smaller than the CFL number.

Public Functions

void **Compute**()

Compute the CFL number. Non-blocking.

float **Get**()

Returns the CFL number. Blocking.

Return cfl number

class Circle : **public** Vortex2D::*Renderer*::*Transformable*, **public** Vortex2D::*Renderer*::*Drawable*

#include <Boundaries.h> Signed distance field of circle.

Public Functions

Circle(**const** *Renderer*::*Device* &*device*, float *radius*, float *extent* = 10.0f)

Initialize the circle with radius and extend of signed distance.

Parameters

- `device`: vulkan device.
- `radius`: radius of circle.
- `extent`: extend how far from the circle the signed distance field is calculated.

void **Initialize**(**const** *Renderer*::*RenderState* &*renderState*)

Initialize the drawable for a particular state. This might include creating the correct pipeline. If it was already initialized, it will do nothing.

Parameters

- `renderState`: the state to initialize with.

void **Update**(**const** glm::mat4 &*projection*, **const** glm::mat4 &*view*)

Update the MVP matrix of the drawable.

Parameters

- `projection`: the projection matrix
- `view`: the view matrix

void **Draw**(vk::CommandBuffer *commandBuffer*, **const** *Renderer*::*RenderState* &*renderState*)

Draw for the given render state. This has to be initialized before.

Parameters

- `commandBuffer`: the command buffer to record into.
- `renderState`: the render state to use.

class ConjugateGradient : **public** Vortex2D::*Fluid*::*LinearSolver*

#include <ConjugateGradient.h> An iterative preconditioned conjugate linear solver. The preconditioner can be specified.

Public Functions

ConjugateGradient(**const** *Renderer*::*Device* &*device*, **const** glm::ivec2 &*size*, *Preconditioner* &*preconditioner*)

Initialize the solver with a size and preconditioner.

Parameters

- `device`: vulkan device
- `size`:
- `preconditioner`:

void **Bind**(*Renderer*::*GenericBuffer* &*d*, *Renderer*::*GenericBuffer* &*l*, *Renderer*::*GenericBuffer* &*b*, *Renderer*::*GenericBuffer* &*x*)

Bind the buffers for the linear solver.

Parameters

- `d`: the diagonal of the matrxi
- `l`: the lower matrix
- `b`: the right hand side
- `x`: the unknowns

void **BindRigidbody**(float *delta*, *Renderer*::*GenericBuffer* &*d*, *RigidBody* &*rigidBody*)

Bind rigidbody with the linear solver's matrix.

Parameters

- `delta`: solver delta
- `d`: diagonal matrix
- `rigidBody`: rigidbody to bind to

void **Solve**(Parameters &*params*, **const** std::vector<*RigidBody* *> &*rigidbodies* = {})

Solve iteratively solve the linear equations in data.

class Density : **public** Vortex2D::*Renderer*::*RenderTexture*, **public** Vortex2D::*Renderer*::*Sprite*

#include <Density.h> *Density* field, used to represent smoke swirling.

class Depth

#include <Multigrid.h> Contains the sizes of the multigrid hierarchy.

Public Functions

Depth(**const** glm::ivec2 &*size*)

Initialize with the finest size.

Parameters

- `size`: the base size.

int **GetMaxDepth**() **const**

The calculated depth of the multigrid.

Return the depth.

glm::ivec2 **GetDepthSize**(std::size_t *i*) **const**
Gets the depth for a given level.

Return the size
Parameters
- `i`: the level

class Diagonal : **public** Vortex2D::*Fluid*::*Preconditioner*
#include <Diagonal.h> *Diagonal* preconditioner. Simplest of preconditioner, useful to verify if the preconditioned conjugate gradient works.

Public Functions

void **Bind**(*Renderer*::*GenericBuffer* &*d*, *Renderer*::*GenericBuffer* &*l*, *Renderer*::*GenericBuffer* &*b*, *Renderer*::*GenericBuffer* &*x*)
Bind the linear equation buffers.

Parameters
- `d`: the diagonal of the matrix
- `l`: the lower matrix
- `b`: the right hand side
- `x`: the unknown buffer

void **Record**(vk::CommandBuffer *commandBuffer*)
Record the preconditioner.

Parameters
- `commandBuffer`: the command buffer to record into.

class DistanceField : **public** Vortex2D::*Renderer*::*AbstractSprite*
#include <Boundaries.h> Sprite of a distance field.

Public Functions

DistanceField(**const** *Renderer*::*Device* &*device*, *Renderer*::*RenderTexture* &*levelSet*, float *scale* = 1.0f)
Initialize the price with the level set and scale.

Parameters
- `device`: vulkan device
- `levelSet`: level set to use as sprite
- `scale`: scale of the level set

void **Draw**(vk::CommandBuffer *commandBuffer*, **const** *Renderer*::*RenderState* &*renderState*)
Draw for the given render state. This has to be initialized before.

Parameters
- `commandBuffer`: the command buffer to record into.
- `renderState`: the render state to use.

class Extrapolation
#include <Extrapolation.h> Class to extrapolate values into the neumann and/or dirichlet boundaries.

Public Functions

void **Extrapolate**()
Will extrapolate values from buffer into the dirichlet and neumann boundaries.

void **ConstrainBind**(*Renderer*::*Texture* &*solidPhi*)
Binds a solid level set to use later and constrain the velocity against.

Parameters
- `solidPhi`: solid level set

void **ConstrainVelocity**()
Constrain the velocity, i.e. ensure that the velocity normal to the solid level set is 0.

class GaussSeidel : **public** Vortex2D::*Fluid*::*LinearSolver*, **public** Vortex2D::*Fluid*::*Preconditioner*
#include <*GaussSeidel.h*> An iterative black and red successive over relaxation linear solver.

Public Functions

void **Bind**(*Renderer*::*GenericBuffer* &*d*, *Renderer*::*GenericBuffer* &*l*, *Renderer*::*GenericBuffer* &*b*, *Renderer*::*GenericBuffer* &*x*)
Bind the buffers for the linear solver.

Parameters
- `d`: the diagonal of the matrxi
- `l`: the lower matrix
- `b`: the right hand side
- `x`: the unknowns

void **BindRigidbody**(float *delta*, *Renderer*::*GenericBuffer* &*d*, *RigidBody* &*rigidBody*)
Bind rigidbody with the linear solver's matrix.

Parameters
- `delta`: solver delta
- `d`: diagonal matrix
- `rigidBody`: rigidbody to bind to

void **Solve**(Parameters &*params*, **const** std::vector<*RigidBody* *> &*rigidbodies* = {})
Iterative solving of the linear equations in data.

void **Record**(vk::CommandBuffer *commandBuffer*)
Record the preconditioner.

Parameters
- `commandBuffer`: the command buffer to record into.

void **Record**(vk::CommandBuffer *commandBuffer*, int *iterations*)
Record a determined number of iterations.

Parameters
- `commandBuffer`:
- `iterations`:

void **SetW**(float *w*)
Set the w factor of the GS iterations : x_new = w * x_new + (1-w) * x_old.

Parameters
- `w`:

void **SetPreconditionerIterations**(int *iterations*)
set number of iterations to be used when GS is a preconditioner

Parameters
- `iterations`:

class IncompletePoisson : **public** Vortex2D::*Fluid*::*Preconditioner*
#include <IncompletePoisson.h> Incomplete poisson preconditioner. Slightly better than a simple diagonal preconditioner.

Public Functions

void **Bind**(*Renderer*::*GenericBuffer* &*d*, *Renderer*::*GenericBuffer* &*l*, *Renderer*::*GenericBuffer* &*b*, *Renderer*::*GenericBuffer* &*x*)
Bind the linear equation buffers.

Parameters
- `d`: the diagonal of the matrix
- `l`: the lower matrix
- `b`: the right hand side
- `x`: the unknown buffer

void **Record**(vk::CommandBuffer *commandBuffer*)
Record the preconditioner.

Parameters
- `commandBuffer`: the command buffer to record into.

class Jacobi : **public** Vortex2D::*Fluid*::*Preconditioner*
#include <Jacobi.h> An iterative jacobi linear solver.

Public Functions

void **Bind**(*Renderer*::*GenericBuffer* &*d*, *Renderer*::*GenericBuffer* &*l*, *Renderer*::*GenericBuffer* &*b*, *Renderer*::*GenericBuffer* &*x*)
Bind the linear equation buffers.

Parameters
- `d`: the diagonal of the matrix
- `l`: the lower matrix
- `b`: the right hand side
- `x`: the unknown buffer

void **Record**(vk::CommandBuffer *commandBuffer*)
Record the preconditioner.

Parameters
- `commandBuffer`: the command buffer to record into.

void **SetW**(float *w*)
Set the w factor of the GS iterations : x_new = w * x_new + (1-w) * x_old.

Parameters
- `w`:

void **SetPreconditionerIterations**(int *iterations*)
set number of iterations to be used when GS is a preconditioner

Parameters
- `iterations`:

class **LevelSet** : **public** Vortex2D::*Renderer*::*RenderTexture*

#include <*LevelSet.h*> A signed distance field, which can be re-initialized. In other words, a level set.

Public Functions

void **Reinitialise**()
Reinitialise the level set, i.e. ensure it is a correct signed distance field.

void **ShrinkWrap**()
Shrink wrap wholes.

void **ExtrapolateBind**(*Renderer*::*Texture* &*solidPhi*)
Bind a solid level set, which will be used to extrapolate into this level set.

Parameters
- `solidPhi`:

void **Extrapolate**()
Extrapolate this level set into the solid level set it was attached to. This only performs a single cell extrapolation.

struct **LinearSolver**

#include <*LinearSolver.h*> An interface to represent a linear solver.

Subclassed by *Vortex2D::Fluid::ConjugateGradient*, *Vortex2D::Fluid::GaussSeidel*

Public Functions

virtual void **Bind**(*Renderer*::*GenericBuffer* &*d*, *Renderer*::*GenericBuffer* &*l*, *Renderer*::*GenericBuffer* &*b*, *Renderer*::*GenericBuffer* &*x*) = 0
Bind the buffers for the linear solver.

Parameters
- `d`: the diagonal of the matrxi
- `l`: the lower matrix
- `b`: the right hand side
- `x`: the unknowns

virtual void **BindRigidbody**(float *delta*, *Renderer*::*GenericBuffer* &*d*, *RigidBody* &*rigidBody*) = 0
Bind rigidbody with the linear solver's matrix.

Parameters
- `delta`: solver delta
- `d`: diagonal matrix
- `rigidBody`: rigidbody to bind to

virtual void **Solve**(*Parameters* &*params*, **const** std::vector<*RigidBody* *> &*rigidBodies* = {}) = 0
Solves the linear equations.

Parameters
- `params`: solver iteration/error parameters
- `rigidBodies`: rigidbody to include in solver's matrix

struct Data
#include <LinearSolver.h> The various parts of linear equations.

struct Parameters
#include <LinearSolver.h> *Parameters* for an iterative linear solvers.

Public Types

enum SolverType
Run the solver a fixed number of step or until we reached a minimum error.

Values:

Fixed

Iterative

Public Functions

Parameters(*SolverType type*, unsigned *iterations*, float *errorTolerance* = 0.0f)
Construct parameters with max iterations and max error.

Parameters
- `type`: fixed or iterative type of solver
- `iterations`: max number of iterations to perform
- `errorTolerance`: solver stops when the error is smaller than this.

bool **IsFinished**(float *initialError*) **const**
Checks if we've reacched the parameters.

Return if we can stop the linear solver.
Parameters
- `initialError`: the initial error

void **Reset**()
Sets the out error and out iterations to 0.

class LocalGaussSeidel : **public** Vortex2D::*Fluid*::*Preconditioner*
#include <GaussSeidel.h> A version of the gauss seidel that can only be applied on sizes (16,16) or smaller.

Public Functions

void **Bind**(*Renderer*::*GenericBuffer* &*d*, *Renderer*::*GenericBuffer* &*l*, *Renderer*::*GenericBuffer* &*b*, *Renderer*::*GenericBuffer* &*x*)
Bind the linear equation buffers.

Parameters
- `d`: the diagonal of the matrix
- `l`: the lower matrix
- `b`: the right hand side
- `x`: the unknown buffer

void **Record**(vk::CommandBuffer *commandBuffer*)
Record the preconditioner.

Parameters
- `commandBuffer`: the command buffer to record into.

class Multigrid : **public** Vortex2D::*Fluid*::*Preconditioner*

#include <Multigrid.h> *Multigrid* preconditioner. It creates a hierarchy of twice as small set of linear equations. It applies a few iterations of jacobi on each level and transfers the error on the level above. It then copies the error down, adds to the current solution and apply a few more iterations of jacobi.

Public Functions

Multigrid(**const** *Renderer*::*Device* &*device*, **const** glm::ivec2 &*size*, float *delta*)

Initialize multigrid for given size and delta.

Parameters
- `device`: vulkan device
- `size`: of the linear equations
- `delta`: timestep delta

void **Bind**(*Renderer*::*GenericBuffer* &*d*, *Renderer*::*GenericBuffer* &*l*, *Renderer*::*GenericBuffer* &*b*, *Renderer*::*GenericBuffer* &*x*)

Bind the linear equation buffers.

Parameters
- `d`: the diagonal of the matrix
- `l`: the lower matrix
- `b`: the right hand side
- `x`: the unknown buffer

void **BuildHierarchiesBind**(*Pressure* &*pressure*, *Renderer*::*Texture* &*solidPhi*, *Renderer*::*Texture* &*liquidPhi*)

Bind the level sets from which the hierarchy is built.

Parameters
- `pressure`: The current linear equations
- `solidPhi`: the solid level set
- `liquidPhi`: the liquid level set

void **BuildHierarchies**()

Computes the hierarchy to be used by the multigrid. Asynchronous operation.

void **Record**(vk::CommandBuffer *commandBuffer*)

Record the preconditioner.

Parameters
- `commandBuffer`: the command buffer to record into.

class ParticleCount : **public** Vortex2D::*Renderer*::*RenderTexture*

#include <Particles.h> Container for particles used in the advection of the fluid simulation. Also a level set that is built from the particles.

Public Functions

void **Scan**()

Count the number of particles and update the internal data structures.

int **GetTotalCount**()

Calculate the total number of particles and return it.

Return

Renderer::*IndirectBuffer*<*Renderer*::*DispatchParams*> &**GetDispatchParams**()
Calculate the dispatch parameters to use on the particle buffer.

Return

void **LevelSetBind**(*LevelSet* &*levelSet*)
Bind a solid level set, which will be used to interpolate the particles out of.

Parameters
- `levelSet`:

void **Phi**()
Calculate the level set from the particles.

void **VelocitiesBind**(*Velocity* &*velocity*, *Renderer*::*GenericBuffer* &*valid*)
Bind the velocities, used for advection of the particles.

Parameters
- `velocity`:
- `valid`:

void **TransferToGrid**()
Interpolate the velocities of the particles to the velocities field.

void **TransferFromGrid**()
Interpolate the velocities field in to the particles' velocity.

class Polygon : **public** Vortex2D::*Renderer*::*Transformable*, **public** Vortex2D::*Renderer*::*Drawable*
#include <*Boundaries.h*> Signed distance field of a poylgon.

Subclassed by *Vortex2D::Fluid::Rectangle*

Public Functions

Polygon(**const** *Renderer*::*Device* &*device*, std::vector<glm::vec2> *points*, bool *inverse* = false, float *extent* = 10.0f)
Initialize polygon with set of points and extent of signed distance.

Parameters
- `device`: vulkan device
- `points`: clockwise oriented set of points (mininum 3).
- `inverse`: flag if the distance field should be inversed.
- `extent`: extend how far from the poylon the signed distance field is calculated.

void **Initialize**(**const** *Renderer*::*RenderState* &*renderState*)
Initialize the drawable for a particular state. This might include creating the correct pipeline. If it was already initialized, it will do nothing.

Parameters
- `renderState`: the state to initialize with.

void **Update**(**const** glm::mat4 &*projection*, **const** glm::mat4 &*view*)
Update the MVP matrix of the drawable.

Parameters
- `projection`: the projection matrix
- `view`: the view matrix

void **Draw**(vk::CommandBuffer *commandBuffer*, **const** *Renderer*::*RenderState* &*renderState*)
Draw for the given render state. This has to be initialized before.

Parameters
- `commandBuffer`: the command buffer to record into.
- `renderState`: the render state to use.

struct Preconditioner
#include <Preconditioner.h> An interface to represent a linear solver preconditioner.

Subclassed by *Vortex2D::Fluid::Diagonal*, *Vortex2D::Fluid::GaussSeidel*, *Vortex2D::Fluid::IncompletePoisson*, *Vortex2D::Fluid::Jacobi*, *Vortex2D::Fluid::LocalGaussSeidel*, *Vortex2D::Fluid::Multigrid*

Public Functions

virtual void **Bind**(*Renderer*::*GenericBuffer* &*d*, *Renderer*::*GenericBuffer* &*l*, *Renderer*::*GenericBuffer* &*b*, *Renderer*::*GenericBuffer* &*x*) = 0
Bind the linear equation buffers.

Parameters
- `d`: the diagonal of the matrix
- `l`: the lower matrix
- `b`: the right hand side
- `x`: the unknown buffer

virtual void **Record**(vk::CommandBuffer *commandBuffer*) = 0
Record the preconditioner.

Parameters
- `commandBuffer`: the command buffer to record into.

class PrefixScan
#include <PrefixScan.h> The prefix sum operator.

```
void PrefixSym(int input[], int n, int output[])
{
    output[0] = input[0];

    for (int i = 1; i < n; i++)
        output[i] = output[i-1] + input[i];
}
```

class Bound
#include <PrefixScan.h> A prefix scan object bound with input/output buffers, ready to be dispatched.

class Pressure
#include <Pressure.h> build the linear equation and compute the divergence from the resulting solution.

Public Functions

Renderer::*Work*::*Bound* **BindMatrixBuild**(**const** glm::ivec2 &*size*, *Renderer*::*GenericBuffer* &*diagonal*, *Renderer*::*GenericBuffer* &*lower*, *Renderer*::*Texture* &*liquidPhi*, *Renderer*::*Texture* &*solidPhi*)
Bind the various buffes for the linear system Ax = b.

Return

Parameters

- `size`: size of the linear system
- `diagonal`: diagonal of A
- `lower`: lower matrix of A
- `liquidPhi`: liquid level set
- `solidPhi`: solid level set

void **BuildLinearEquation**()

Build the matrix A and right hand side b.

void **ApplyPressure**()

Apply the solution of the equation Ax = b, i.e. the pressure to the velocity to make it non-divergent.

class Rectangle : **public** Vortex2D::*Fluid*::*Polygon*

#include <*Boundaries.h*> Signed distance field of a rectangle.

Public Functions

Rectangle(**const** *Renderer*::*Device* &*device*, **const** glm::vec2 &*size*, bool *inverse* = false, float *extent* = 10.0f)

Initialize rectangle with size and extend of signed distance.

Parameters

- `device`: vulkan device.
- `size`: rectangle size
- `inverse`: flag if the distance field should be inverted.
- `extent`: extent how far from the rectangle the signed distance field is calculated.

void **Initialize**(**const** *Renderer*::*RenderState* &*renderState*)

Initialize the drawable for a particular state. This might include creating the correct pipeline. If it was already initialized, it will do nothing.

Parameters

- `renderState`: the state to initialize with.

void **Update**(**const** glm::mat4 &*projection*, **const** glm::mat4 &*view*)

Update the MVP matrix of the drawable.

Parameters

- `projection`: the projection matrix
- `view`: the view matrix

void **Draw**(vk::CommandBuffer *commandBuffer*, **const** *Renderer*::*RenderState* &*renderState*)

Draw for the given render state. This has to be initialized before.

Parameters

- `commandBuffer`: the command buffer to record into.
- `renderState`: the render state to use.

class Reduce

#include <*Reduce.h*> Parallel reduction of a buffer into one value. The operator and type of data is specified by inheriting the class.

Subclassed by *Vortex2D::Fluid::ReduceJ*, *Vortex2D::Fluid::ReduceMax*, *Vortex2D::Fluid::ReduceSum*

Public Functions

Reduce::*Bound* **Bind**(*Renderer*::*GenericBuffer* &*input*, *Renderer*::*GenericBuffer* &*output*)
Bind the reduce operation.

Return a bound object that can be recorded in a command buffer.
Parameters
- `input`: input buffer
- `output`: output buffer

class Bound
#include <Reduce.h> *Bound* input and output buffer for a reduce operation.

Public Functions

void **Record**(vk::CommandBuffer *commandBuffer*)
Record the reduce operation.

Parameters
- `commandBuffer`: the command buffer to record into.

class ReduceJ : **public** Vortex2D::*Fluid*::*Reduce*
#include <Reduce.h> *Reduce* operation on a struct with a 2d vector and 1 float (i.e. 3 floats) with addition.

Public Functions

ReduceJ(**const** *Renderer*::*Device* &*device*, **const** glm::ivec2 &*size*)
Initialize reduce with device and 2d size.

Parameters
- `device`:
- `size`:

class ReduceMax : **public** Vortex2D::*Fluid*::*Reduce*
#include <Reduce.h> *Reduce* operation on float with max of absolute.

Public Functions

ReduceMax(**const** *Renderer*::*Device* &*device*, **const** glm::ivec2 &*size*)
Initialize reduce with device and 2d size.

Parameters
- `device`:
- `size`:

class ReduceSum : **public** Vortex2D::*Fluid*::*Reduce*
#include <Reduce.h> *Reduce* operation on float with addition.

Public Functions

ReduceSum(**const** *Renderer*::*Device* &*device*, **const** glm::ivec2 &*size*)
Initialize reduce with device and 2d size.

Parameters

- `device`:
- `size`:

class RigidBody : **public** Vortex2D::*Renderer*::*Transformable*

#include <Rigidbody.h> Rigidbody that can interact with the fluid: either be push by it, or influence it, or both.

Public Functions

virtual void **ApplyForces**()

function to override and apply forces from this rigidbody to the external rigidbody

virtual void **ApplyVelocities**()

Override and apply velocities from the external rigidbody to the this rigidbody.

void **SetMassData**(float *mass*, float *inertia*)

Sets the mass and inertia of the rigidbody.

Parameters

- `mass`: of the body
- `inertia`: of the body

void **SetVelocities**(**const** glm::vec2 &*velocity*, float *angularVelocity*)

sets the velocities and angular velocities of the body

Parameters

- `velocity`:
- `angularVelocity`:

void **UpdatePosition**()

Upload the transform matrix to the GPU.

void **RenderPhi**()

Render the current object orientation in an internal texture and the external one.

void **BindPhi**(*Renderer*::*RenderTexture* &*phi*)

Bind the rendertexture where this rigidbodies shape will be rendered.

Parameters

- `phi`: render texture of the world

void **BindDiv**(*Renderer*::*GenericBuffer* &*div*, *Renderer*::*GenericBuffer* &*diagonal*)

Bind a the right hand side and diagonal of the linear system Ax = b. This is to apply the rigid body influence to the system.

Parameters

- `div`: right hand side of the linear system Ax=b
- `diagonal`: diagonal of matrix A

void **BindVelocityConstrain**(*Fluid*::*Velocity* &*velocity*)

Bind velocities to constrain based on the body's velocity.

Parameters

- `velocity`:

void **BindForce**(*Renderer*::*GenericBuffer* &*d*, *Renderer*::*GenericBuffer* &*pressure*)

Bind pressure, to have the pressure update the body's forces.

Parameters

- `d`: diagonal of matrix A
- `pressure`: solved pressure buffer

void **BindPressure**(float *delta*, *Renderer*::*GenericBuffer* &*d*, *Renderer*::*GenericBuffer* &*s*, *Renderer*::*GenericBuffer* &*z*)
Bind pressure, to have the pressure update the body's forces.

Parameters
- `delta`:
- `d`:
- `s`:
- `z`:

void **Div**()
Apply the body's velocities to the linear equations matrix A and right hand side b.

void **Force**()
Apply the pressure to body, updating its forces.

void **Pressure**()
Reduce the force for pressure update.

void **VelocityConstrain**()
Constrain the velocities field based on the body's velocity.

Velocity **GetForces**()
Download the forces from the GPU and return them.

Return

vk::Flags<Type> **GetType**()
Type of this body.

Return

void **SetType**(Type *type*)
Set the type of the body.

Parameters
- `type`:

Renderer::*RenderTexture* &**Phi**()
the local level set of the body

Return

class RigidBodySolver
#include <*Rigidbody.h*> Interface to call the external rigidbody solver.

Public Functions

virtual void **Step**(float *delta*) = 0
perfoms a single step of the solver.

Parameters
- `delta`: of simulation

class SmokeWorld : **public** Vortex2D::*Fluid*::*World*
#include <*World.h*> A concrete implementation of *World* to simulate 'smoke', or more accurately dye in a liquid. The liquid cannot change location or size.

Public Functions

void **FieldBind**(*Density* &*density*)
Bind a density field to be moved around with the fluid.

Parameters
- `density`: the density field

class Transfer
#include <Transfer.h> Prolongates or restrict a level set on a finer or coarser level set.

Public Functions

Transfer(**const** *Renderer*::*Device* &*device*)
Initialize prolongate and restrict compute pipelines.

Parameters
- `device`:

void **ProlongateBind**(std::size_t *level*, **const** glm::ivec2 &*fineSize*, *Renderer*::*GenericBuffer* &*fine*, *Renderer*::*GenericBuffer* &*fineDiagonal*, *Renderer*::*GenericBuffer* &*coarse*, *Renderer*::*GenericBuffer* &*coarseDiagonal*)
Prolongate a level set on a finer level set. Setting the 4 cells to the value of the coarser grid. Multiple level sets can be bound and indexed.

Parameters
- `level`: the index of the bound level set to prolongate
- `fineSize`: size of the finer level set
- `fine`: the finer level set
- `fineDiagonal`: the diagonal of the linear equation matrix at size `fineSize`
- `coarse`: the coarse level set
- `coarseDiagonal`: the diagonal of the linear equation matrix at size half of `fineSize`

void **RestrictBind**(std::size_t *level*, **const** glm::ivec2 &*fineSize*, *Renderer*::*GenericBuffer* &*fine*, *Renderer*::*GenericBuffer* &*fineDiagonal*, *Renderer*::*GenericBuffer* &*coarse*, *Renderer*::*GenericBuffer* &*coarseDiagonal*)
Restricing the level set on a coarser level set. Averages 4 cells into one. Multiple level sets can be bound and indexed.

Parameters
- `level`: the index of the bound level set to prolongate
- `fineSize`: size of the finer level set
- `fine`: the finer level set
- `fineDiagonal`: the diagonal of the linear equation matrix at size `fineSize`
- `coarse`: the coarse level set
- `coarseDiagonal`: the diagonal of the linear equation matrix at size half of `fineSize`

void **Prolongate**(vk::CommandBuffer *commandBuffer*, std::size_t *level*)
Prolongate the level set, using the bound level sets at the specified index.

Parameters
- `commandBuffer`: command buffer to record into.
- `level`: index of bound level sets.

void **Restrict**(vk::CommandBuffer *commandBuffer*, std::size_t *level*)
Restrict the level set, using the bound level sets at the specified index.

Parameters

- `commandBuffer`: command buffer to record into.
- `level`: index of bound level sets.

class Velocity : **public** Vortex2D::*Renderer*::*RenderTexture*

#include <Velocity.h> The *Velocity* field. Can be used to calculate a difference between different states. Contains three fields: intput and output, used for ping-pong algorithms, and d, the difference between two velocity fields.

Public Functions

Renderer::*Texture* &**Output**()

An output texture used for algorithms that used the velocity as input and need to create a new velocity field.

Return

Renderer::*Texture* &**D**()

A difference velocity field, calculated with the difference between this velocity field, and the output velocity field.

Return

void **CopyBack**(vk::CommandBuffer *commandBuffer*)

Copy the output field to the main field.

Parameters

- `commandBuffer`:

void **Clear**(vk::CommandBuffer *commandBuffer*)

Clear the velocity field.

Parameters

- `commandBuffer`:

void **SaveCopy**()

Copy to the difference field.

void **VelocityDiff**()

Calculate the difference between the difference field and this velocity field, store it in the diference field.

class WaterWorld : **public** Vortex2D::*Fluid*::*World*

#include <World.h> A concrete implementation of *World* to simulate water.

Public Functions

Renderer::*RenderCommand* **RecordParticleCount**(*Renderer*::*RenderTarget*::DrawableList *drawables*)

The water simulation uses particles to define the water area. In fact, the level set is built from the particles. This means to be able to set an area, we can't use *RecordLiquidPhi*. To define the particle area, simply draw a regular shape. The colour r is used to determine if we add or remove particles, use r = 4 to add and r = -4 to remove.

Return render command

Parameters

- `drawables`: list of drawables object with colour 8 or -8

void **ParticlePhi**()
Using the particles, create a level set (phi) encompassing all the particles. This can be viewed with *LiquidDistanceField*.

class World
#include <World.h> The main class of the framework. Each instance manages a grid and this class is used to set forces, define boundaries, solve the incompressbility equations and do the advection.

Subclassed by *Vortex2D::Fluid::SmokeWorld*, *Vortex2D::Fluid::WaterWorld*

Public Functions

World(**const** *Renderer*::*Device* &*device*, **const** glm::ivec2 &*size*, float *dt*, int *numSubSteps* = 1)
Construct an Engine with a size and time step.

Parameters
- `device`: vulkan device
- `size`: dimensions of the simulation
- `dt`: timestamp of the simulation, e.g. 0.016 for 60FPS simulations.
- `numSubSteps`: the number of sub-steps to perform per step call. Reduces loss of fluid.

void **Step**(*LinearSolver*::*Parameters* &*params*)
Perform one step of the simulation.

Renderer::*RenderCommand* **RecordVelocity**(*Renderer*::*RenderTarget*::DrawableList *drawables*, *VelocityOp op*)
Record drawables to the velocity field. The colour (r,g) will be used as the velocity (x, y)

Return render command
Parameters
- `drawables`: a list of drawable field
- `op`: operation of the render: add velocity or set velocity

void **SubmitVelocity**(*Renderer*::*RenderCommand* &*renderCommand*)
submit the render command created with *RecordVelocity*

Parameters
- `renderCommand`: the render command

Renderer::*RenderCommand* **RecordLiquidPhi**(*Renderer*::*RenderTarget*::DrawableList *drawables*)
Record drawables to the liquid level set, i.e. to define the fluid area. The drawables need to make a signed distance field, if not the result is undefined.

Return render command
Parameters
- `drawables`: a list of signed distance field drawables

Renderer::*RenderCommand* **RecordStaticSolidPhi**(*Renderer*::*RenderTarget*::DrawableList *drawables*)
Record drawables to the solid level set, i.e. to define the boundary area. The drawables need to make a signed distance field, if not the result is undefined.

Return render command
Parameters
- `drawables`: a list of signed distance field drawables

DistanceField **LiquidDistanceField**()
Create sprite that can be rendered to visualize the liquid level set.

Return a sprite

DistanceField **SolidDistanceField**()
Create sprite that can be rendered to visualize the solid level set.

Return a sprite

void **AddRigidbody**(*RigidBody* &*rigidbody*)
Add a rigibody to the solver.

Parameters
- `rigidbody`:

void **RemoveRigidBody**(*RigidBody* &*rigidbody*)
Remove a rigidbody from the solver.

Parameters
- `rigidbody`:

void **AttachRigidBodySolver**(*RigidBodySolver* &*rigidbodySolver*)
Attach a rigidbody solver, e.g. box2d.

Parameters
- `rigidbodySolver`:

float **GetCFL**()
Calculate the CFL number, i.e. the width divided by the max velocity.

Return CFL number

Renderer::*Texture* &**GetVelocity**()
Get the velocity, can be used to display it.

Return velocity field reference

Index

V