Vortex2D Documentation

Release 1.0

Maximilian Maldacker

May 02, 2022

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

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

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

Vortex::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 *Vortex::Renderer::RenderCommand* on a *Vortex::Renderer::RenderTarget*. There are two implementations of it:

- *Vortex::Renderer::RenderWindow*
- *Vortex::Renderer::RenderTexture*

You can render implementations of the abstract class *Vortex::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 *Vortex::Renderer::ColorBlendState*.

Shapes

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

```
Vortex::Renderer::Rectangle rectangle(device, {100.0f, 100.0f});
Vortex::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.

```
Vortex::Renderer::Texture texture(device, 100, 100, vk::Format::eR8G8B8A8Unorm);
Vortex::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:

```
Vortex::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 *`Vortex::Fluid::LevelSet`* inherits *`Vortex::Renderer::RenderTexture`*.

Basic shapes

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

- *`Vortex::Fluid::Rectangle`*
- *`Vortex::Fluid::Polygon`*
- *`Vortex::Fluid::Circle`*

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

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

Vortex::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:

> - `Vortex::Renderer::IntersectionBlend`
> - `Vortex::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 `Vortex::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 `Vortex::Renderer::RenderCommand::Submit()` we pass the *`Vortex::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 `Vortex::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 *`Vortex::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:

- `Vortex::Fluid::RigidBody::ApplyForces()`
- `Vortex::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() & Vortex::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() & Vortex::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 `Vortex::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

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

API Reference

namespace Renderer

Typedefs

using DrawablePtr = std::shared_ptr<*Drawable*>

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

Enums

enum MemoryUsage

Values:

Gpu

Cpu

CpuToGpu

GpuToCpu

enum Format

Values:

R8Uint

R8Sint

R32Sfloat

R32Sint

R8G8B8A8Unorm

B8G8R8A8Unorm

R32G32Sfloat

R32G32B32A32Sfloat

enum ShaderStage

Values:

Vertex

Fragment

Compute

enum PrimitiveTopology

Values:

Triangle

LineList

enum BufferUsage

Values:

Vertex

Uniform

Storage

Indirect

Index

enum Access

Values:

None

Write

Read

enum ImageLayout

Values:

General

enum PipelineBindPoint

Values:

Graphics

Compute

enum BlendFactor

Values:

Zero

One

ConstantColor

SrcAlpha

OneMinusSrcAlpha

enum BlendOp

Values:

Add

Max

Min

enum BindType

Values:

StorageBuffer

StorageImage

ImageSampler

UniformBuffer

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.

bool **operator==**(const *GraphicsPipelineDescriptor*::ShaderDescriptor &*left*, const *GraphicsPipelineDescriptor*::ShaderDescriptor &*right*)

bool **operator==**(const *GraphicsPipelineDescriptor*::VertexBindingDescriptor &*left*, const *GraphicsPipelineDescriptor*::VertexBindingDescriptor &*right*)

bool **operator==**(const *GraphicsPipelineDescriptor*::VertexAttributeDescriptor &*left*, const *GraphicsPipelineDescriptor*::VertexAttributeDescriptor &*right*)

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

bool **operator==**(const *SpecConstInfo*::Entry &*left*, const *SpecConstInfo*::Entry &*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*)

VORTEX_API std::uint64_t Vortex::Renderer::GetBytesPerPixel(Format format)

Gets the number of bytes per pixel given the format.

Return bytes per pixel

Parameters

- `format`: of texture

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

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

VmaMemoryUsage **ConvertMemoryUsage**(*MemoryUsage memoryUsage*)

vk::Format **ConvertFormat**(*Format format*)

vk::ShaderStageFlagBits **ConvertShaderStage**(*ShaderStage shaderStage*)

vk::PrimitiveTopology **ConvertTopology**(*PrimitiveTopology topology*)

vk::BufferUsageFlags **ConvertBufferUsage**(*BufferUsage bufferUsage*)

vk::ImageLayout **ConvertImageLayout**(*ImageLayout layout*)

vk::AccessFlags **ConvertAccess**(*Access access*)

vk::PipelineBindPoint **ConvertPipelineBindPoint**(*PipelineBindPoint bindPoint*)

vk::BlendFactor **ConvertBlendFactor**(*BlendFactor factor*)

vk::BlendOp **ConvertBlendOp**(*BlendOp blendOp*)

vk::DescriptorType **ConvertDescriptorType**(*BindType type*)

VORTEX_API ComputeSize Vortex::Renderer::MakeStencilComputeSize(const glm::ivec2 & siz

Create a *ComputeSize* for a stencil type shader.

Return calculate *ComputeSize*

Parameters

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

VORTEX_API ComputeSize Vortex::Renderer::MakeCheckerboardComputeSize(const glm::ivec2

Create a *ComputeSize* for a checkerboard type shader.

Return calculate *ComputeSize*

Parameters

- `size`: the domain size

class AbstractShape : public Vortex::*Renderer*::*Shape*

#include <Shapes.h> An polygonal shape where the fragment shader can be specified for customisation.

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

Public Functions

VORTEX_API void Vortex::Renderer::AbstractShape::Initialize(const RenderState & r

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.

VORTEX_API void Vortex::Renderer::AbstractShape::Update(const glm::mat4 & projectio

Update the MVP matrix of the drawable.

Parameters

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

VORTEX_API void Vortex::Renderer::AbstractShape::Draw(CommandEncoder & commandEnco

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 Vortex::*Renderer*::*Drawable*, **public** Vortex::*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 *Vortex::Fluid::DistanceField*, *Vortex::Renderer::Sprite*

Public Functions

VORTEX_API void Vortex::Renderer::AbstractSprite::Initialize(const 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.

VORTEX_API void Vortex::Renderer::AbstractSprite::Update(const glm::mat4 & projecti

Update the MVP matrix of the drawable.

Parameters

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

VORTEX_API void Vortex::Renderer::AbstractSprite::Draw(CommandEncoder & commandEnc

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 <BindGroup.h> The texture/sampler or buffer that can be binded to a shader.

template<typename **T**>
class Buffer : **public** Vortex::*Renderer*::*GenericBuffer*

#include <Buffer.h> a storage buffer type of buffer

class Clear : **public** Vortex::*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**(CommandEncoder &*commandEncoder*, **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

VORTEX_API **`CommandBuffer`**(*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.

`VORTEX_API CommandBuffer& Vortex::Renderer::CommandBuffer::Record(CommandFn comman`

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.

`VORTEX_API CommandBuffer& Vortex::Renderer::CommandBuffer::Record(const RenderTarg`

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.

`VORTEX_API CommandBuffer& Vortex::Renderer::CommandBuffer::Wait()`

Wait for the command submit to finish. Does nothing if the synchronise flag was false.

`VORTEX_API CommandBuffer& Vortex::Renderer::CommandBuffer::Reset()`

Reset the command buffer so it can be recorded again.

`VORTEX_API CommandBuffer& Vortex::Renderer::CommandBuffer::Submit(const std::initia`

submit the command buffer

VORTEX_API **`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

VORTEX_API **`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

VORTEX_API **`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 VORTEX_API glm::ivec2 Vortex::Renderer::ComputeSize::GetLocalSize2D()

The default local size for 2D compute shaders.

Return a 2d vector

static VORTEX_API int Vortex::Renderer::ComputeSize::GetLocalSize1D()

The default local size for 1D compute shaders.

Return a integer value

static VORTEX_API glm::ivec2 Vortex::Renderer::ComputeSize::GetWorkSize(const glm::

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 VORTEX_API glm::ivec2 Vortex::Renderer::ComputeSize::GetWorkSize(int size, i

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 VORTEX_API ComputeSize Vortex::Renderer::ComputeSize::Default2D()

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

Return a default compute size

static VORTEX_API ComputeSize Vortex::Renderer::ComputeSize::Default1D()

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

Return a default compute size

class Device

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

Subclassed by *Vortex::Renderer::VulkanDevice*

Public Functions

virtual VORTEX_API Handle::BindGroupLayout Vortex::Renderer::Device::CreateBindGrou

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

Return cached descriptor set layout

Parameters

- `layout`: pipeline layout

virtual VORTEX_API Handle::PipelineLayout Vortex::Renderer::Device::CreatePipelineI

create, cache and return a vulkan pipeline layout given the layout

Return vulkan pipeline layout

Parameters

- `layout`: pipeline layout

virtual VORTEX_API Handle::Pipeline Vortex::Renderer::Device::CreateGraphicsPipelin

Create a graphics pipeline.

Return

Parameters

- `builder`:
- `renderState`:

virtual VORTEX_API Handle::Pipeline Vortex::Renderer::Device::CreateComputePipeline

Create a compute pipeline.

Parameters

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

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 *Vortex::Fluid::Circle*, *Vortex::Fluid::Polygon*, *Vortex::Renderer::AbstractSprite*, *Vortex::Renderer::Clear*, *Vortex::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**(CommandEncoder &*commandEncoder*, **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 : **public** DispatchLoaderBase

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

class Ellipse : **public** Vortex::*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**(CommandEncoder &*commandEncoder*, **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 *Vortex::Renderer::Buffer< float >*, *Vortex::Renderer::Buffer< glm::ivec2 >*, *Vortex::Renderer::Buffer< glm::vec2 >*, *Vortex::Renderer::Buffer< int >*, *Vortex::Renderer::Buffer< std::uint32_t >*, *Vortex::Renderer::Buffer< Vortex::Fluid::Contour::Voxel >*, *Vortex::Renderer::Buffer< Vortex::Fluid::Particle >*, *Vortex::Renderer::Buffer< Vortex::Fluid::RigidBody::Velocity >*, *Vortex::Renderer::Buffer< Vortex::Renderer::DispatchParams >*, *Vortex::Renderer::Buffer< Vortex::Renderer::DrawIndexedIndirect >*, *Vortex::Renderer::IndexBuffer< std::uint32_t >*, *Vortex::Renderer::IndirectBuffer< Vortex::Renderer::DispatchParams >*, *Vortex::Renderer::UniformBuffer< glm::mat4 >*, *Vortex::Renderer::UniformBuffer< glm::vec2 >*, *Vortex::Renderer::UniformBuffer< glm::vec4 >*, *Vortex::Renderer::UniformBuffer< Size >*, *Vortex::Renderer::UniformBuffer< Vortex::Fluid::RigidBody::Velocity >*, *Vortex::Renderer::VertexBuffer< glm::vec2 >*, *Vortex::Renderer::VertexBuffer< Vortex::Renderer::AbstractSprite::Vertex >*, *Vortex::Renderer::Buffer< T >*, *Vortex::Renderer::IndexBuffer< T >*, *Vortex::Renderer::IndirectBuffer< T >*, *Vortex::Renderer::UniformBuffer< T >*, *Vortex::Renderer::VertexBuffer< T >*

Public Functions

VORTEX_API void Vortex::Renderer::GenericBuffer::CopyFrom(CommandEncoder & commanc

Copy a buffer to this buffer.

Parameters

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

VORTEX_API void Vortex::Renderer::GenericBuffer::CopyFrom(CommandEncoder & commanc

Copy a texture to this buffer.

Parameters

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

VORTEX_API Handle::Buffer Vortex::Renderer::GenericBuffer::Handle() const

The vulkan handle.

`VORTEX_API std::uint64_t Vortex::Renderer::GenericBuffer::Size() const`

The size in bytes of the buffer.

`VORTEX_API void Vortex::Renderer::GenericBuffer::Resize(std::uint64_t size)`

Resize the buffer. Invalidates the buffer handle.

Parameters

- `size`: buffer size

`VORTEX_API void Vortex::Renderer::GenericBuffer::Barrier(CommandEncoder & command,`

Inserts a barrier for this buffer.

Parameters

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

`VORTEX_API void Vortex::Renderer::GenericBuffer::Clear(CommandEncoder & command)`

Clear the buffer with 0.

Parameters

- `commandBuffer`: the command buffer to clear on

`VORTEX_API void Vortex::Renderer::GenericBuffer::CopyFrom(uint32_t offset, const vo`

copy from data to buffer

Parameters

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

`VORTEX_API void Vortex::Renderer::GenericBuffer::CopyTo(uint32_t offset, void * dat`

copy buffer to data

Parameters

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

`class GraphicsPipelineDescriptor`

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

Public Functions

`VORTEX_API GraphicsPipelineDescriptor& Vortex::Renderer::GraphicsPipelineDescriptor`

Set the shader.

Return *this

Parameters

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

`VORTEX_API GraphicsPipelineDescriptor& Vortex::Renderer::GraphicsPipelineDescriptor`

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

`VORTEX_API GraphicsPipelineDescriptor& Vortex::Renderer::GraphicsPipelineDescriptor`

Sets the vertex binding.

Return *this

Parameters

- `binding`: binding in the shader
- `stride`: stride in bytes

`struct Image`

#include <BindGroup.h> The texture or sampler that can be bound to a shader.

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

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

template<typename **T**>
`class IndirectBuffer` : **public** Vortex::*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** Vortex::*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 Mesh` : **public** Vortex::*Renderer*::*Shape*

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

Public Functions

`VORTEX_API void Vortex::Renderer::Mesh::Initialize(const RenderState & renderStat`

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.

`VORTEX_API void Vortex::Renderer::Mesh::Update(const glm::mat4 & projection, const`

Update the MVP matrix of the drawable.

Parameters

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

`VORTEX_API void Vortex::Renderer::Mesh::Draw(CommandEncoder & commandEncoder, cons`

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 Rectangle` : **public** Vortex::*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

VORTEX_API RenderCommand& Vortex::Renderer::RenderCommand::Submit(const glm::mat4 &

Submit the render command with a transform matrix.

Return *this

Parameters

- `view`: a transform matrix

VORTEX_API void Vortex::Renderer::RenderCommand::Wait()

Wait for the render command to complete.

VORTEX_API **operator bool**() **const**

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

class RenderpassBuilder

#include <RenderPass.h> Factory for a vulkan render pass.

Public Functions

RenderpassBuilder &**Attachement**(*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::RenderPass **`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 *Vortex::Renderer::RenderTexture*, *Vortex::Renderer::RenderWindow*

class RenderTexture : **public** Vortex::*Renderer*::*RenderTarget*, **public** Vortex::*Renderer*::*Texture*

#include <RenderTexture.h> A render target that renders into a texture.

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

class RenderWindow : **public** Vortex::*Renderer*::*RenderTarget*

#include <RenderWindow.h> Render to a swapchain, i.e. to the window/surface.

Public Functions

VORTEX_API **RenderWindow**(*Device* &*device*, *Handle*::*Surface* *surface*, uint32_t *width*, uint32_t *height*)

Initialize with a given surface and size.

Parameters

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

VORTEX_API void Vortex::Renderer::RenderWindow::Display()

Submits all the render command and present the surface for display.

class Shape : **public** Vortex::*Renderer*::*Drawable*, **public** Vortex::*Renderer*::*Transformable*

#include <Shapes.h> *Shape* interface which is drawable, transformable and has a color.

Subclassed by *Vortex::Renderer::AbstractShape*, *Vortex::Renderer::Ellipse*, *Vortex::Renderer::Mesh*

struct SpecConstInfo

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

class SpirvBinary

#include <Common.h> A binary SPIRV shader, to be feed to vulkan.

class Sprite : **public** Vortex::*Renderer*::*AbstractSprite*

#include <Sprite.h> A sprite that renders a texture with a simple pass-through fragment shader.

Subclassed by *Vortex::Fluid::Density*

class Texture

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

Subclassed by *Vortex::Renderer::RenderTexture*

Public Functions

VORTEX_API void Vortex::Renderer::Texture::CopyFrom(const void * data)

Copies width*heigh*bytesPerPixel amount of data.

Parameters

- `data`: source data

VORTEX_API void Vortex::Renderer::Texture::CopyTo(void * data)

Copies width*heigh*bytesPerPixel amount of data.

Parameters

- `data`: destination data

VORTEX_API void Vortex::Renderer::Texture::CopyFrom(CommandEncoder & command, Text

Copies source texture in this texture.

Parameters

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

VORTEX_API void Vortex::Renderer::Texture::Barrier(CommandEncoder & command, Image

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

class Timer

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

Public Functions

VORTEX_API void Vortex::Renderer::Timer::Start(CommandEncoder & command)

Start the timer after the current last command buffer.

Parameters

- `commandBuffer`: command buffer to write timestamp

VORTEX_API void Vortex::Renderer::Timer::Stop(CommandEncoder & command)

Start the timer after the current last command buffer.

Parameters

- `commandBuffer`: command buffer to write timestamp

VORTEX_API void Vortex::Renderer::Timer::Start()

Start the timer after the current last command buffer.

VORTEX_API void Vortex::Renderer::Timer::Stop()

Stop the timer after the current last command buffer.

VORTEX_API void Vortex::Renderer::Timer::Wait()

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

VORTEX_API uint64_t Vortex::Renderer::Timer::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 *Vortex::Fluid::Circle*, *Vortex::Fluid::Polygon*, *Vortex::Fluid::RigidBody*, *Vortex::Renderer::AbstractSprite*, *Vortex::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** Vortex::*Renderer*::*GenericBuffer*

#include <Buffer.h> a uniform buffer type of buffer

template<typename **T**>
class VertexBuffer : **public** Vortex::*Renderer*::*GenericBuffer*

#include <Buffer.h> a vertex buffer type of buffer

class VulkanDevice : **public** Vortex::*Renderer*::*Device*

Public Functions

Handle::*BindGroupLayout* **CreateBindGroupLayout**(**const** SPIRV::ShaderLayouts &*layout*)

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

Return cached descriptor set layout

Parameters

- `layout`: pipeline layout

Handle::*PipelineLayout* **CreatePipelineLayout**(**const** SPIRV::ShaderLayouts &*layout*)

create, cache and return a vulkan pipeline layout given the layout

Return vulkan pipeline layout

Parameters

- `layout`: pipeline layout

Handle::*Pipeline* **CreateGraphicsPipeline**(**const** *GraphicsPipelineDescriptor* &*builder*, **const** *RenderState* &*renderState*)

Create a graphics pipeline.

Return

Parameters

- `builder`:
- `renderState`:

Handle::*Pipeline* **CreateComputePipeline**(*Handle*::*ShaderModule shader*, *Handle*::*PipelineLayout layout*, *SpecConstInfo specConstInfo* = {})

Create a compute pipeline.

Parameters

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

class Work

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

Public Functions

VORTEX_API **Work**(*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

VORTEX_API Bound Vortex::Renderer::Work::Bind(const std::vector< BindingInput > &

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

VORTEX_API Bound Vortex::Renderer::Work::Bind(ComputeSize computeSize, const std::v

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**(CommandEncoder *&command*, *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.

VORTEX_API void Vortex::Renderer::Work::Bound::Record(CommandEncoder & command)

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.

VORTEX_API void Vortex::Renderer::Work::Bound::RecordIndirect(CommandEncoder &

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*&)

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

namespace Handle

Typedefs

typedef struct PipelineLayout_T ***PipelineLayout**

typedef struct BindGroupLayout_T ***BindGroupLayout**

typedef struct Pipeline_T ***Pipeline**

typedef struct ShaderModule_T ***ShaderModule**

typedef struct RenderPass_T ***RenderPass**

typedef struct Framebuffer_T ***Framebuffer**

typedef struct CommandBuffer_T ***CommandBuffer**

typedef struct Semaphore_T ***Semaphore**

typedef struct BindGroup_T ***BindGroup**

typedef struct Buffer_T ***Buffer**

typedef struct Sampler_T ***Sampler**

typedef struct Image_T ***Image**

typedef struct ImageView_T ***ImageView**

typedef struct Surface_T ***Surface**

Functions

Semaphore **ConvertSemaphore** (vk::Semaphore *semaphore*)

vk::Semaphore **ConvertSemaphore** (*Semaphore semaphore*)

Framebuffer **ConvertFramebuffer** (vk::Framebuffer *framebuffer*)

vk::CommandBuffer **ConvertCommandBuffer** (*CommandBuffer commandBuffer*)

vk::Image **ConvertImage** (*Image image*)

Image **ConvertImage** (vk::Image *image*)

vk::ImageView **ConvertImageView** (*ImageView imageView*)

ImageView **ConvertImageView** (vk::ImageView *imageView*)

vk::Buffer **ConvertBuffer** (*Buffer buffer*)

vk::Sampler **ConvertSampler** (*Sampler sampler*)

vk::SurfaceKHR **ConvertSurface** (*Surface surface*)

Surface **ConvertSurface** (vk::SurfaceKHR *surface*)

1.2.7 Fluid API reference

Classes

- *Vortex::Fluid::Advection*
- *Vortex::Fluid::Circle*
- *Vortex::Fluid::ConjugateGradient*
- *Vortex::Fluid::Density*
- *Vortex::Fluid::Depth*
- *Vortex::Fluid::Diagonal*
- *Vortex::Fluid::DistanceField*
- *Vortex::Fluid::Extrapolation*
- *Vortex::Fluid::GaussSeidel*
- *Vortex::Fluid::IncompletePoisson*
- *Vortex::Fluid::Jacobi*
- *Vortex::Fluid::LevelSet*
- *Vortex::Fluid::LinearSolver*
- *Vortex::Fluid::LocalGaussSeidel*
- *Vortex::Fluid::Multigrid*
- *Vortex::Fluid::ParticleCount*
- *Vortex::Fluid::Polygon*

- *Vortex::Fluid::Preconditioner*
- *Vortex::Fluid::Pressure*
- *Vortex::Fluid::Rectangle*
- *Vortex::Fluid::Reduce*
- *Vortex::Fluid::ReduceJ*
- *Vortex::Fluid::ReduceMax*
- *Vortex::Fluid::ReduceSum*
- *Vortex::Fluid::RigidBody*
- *Vortex::Fluid::SmokeWorld*
- *Vortex::Fluid::Transfer*
- *Vortex::Fluid::Velocity*
- *Vortex::Fluid::WaterWorld*
- *Vortex::Fluid::World*

API Reference

namespace Fluid

Enums

enum VelocityOp

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

Values:

Add

Set

Functions

VORTEX_API LinearSolver::Parameters Vortex::Fluid::FixedParams(unsigned iterations)

Create a linear solver parameters object with fixed solver type.

Return parameters

Parameters

- `iterations`: number of iterations to do

VORTEX_API LinearSolver::Parameters Vortex::Fluid::IterativeParams(float errorToleranc

Create a linear solver parameters object, solver will continue until error tolerance is reached.

Return parameters

Parameters

- `errorTolerance`: tolerance to reach before exiting

VORTEX_API float Vortex::Fluid::DefaultParticleSize()

Variables

VORTEX_API Renderer::ColorBlendState Vortex::Fluid::IntersectionBlend

VORTEX_API Renderer::ColorBlendState Vortex::Fluid::UnionBlend

VORTEX_API std::shared_ptr<Renderer::Clear> Vortex::Fluid::BoundariesClear

class Advection

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

Public Functions

VORTEX_API **Advection** (*Renderer*::*Device* &*device*, **const** glm::ivec2 &*size*, float *dt*, *Velocity* &*velocity*, *Velocity*::InterpolationMode *interpolationMode*)

Initialize advection kernels and related object.

Parameters

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

VORTEX_API void Vortex::Fluid::Advection::AdvectVelocity()

Self advect velocity.

VORTEX_API void Vortex::Fluid::Advection::AdvectBind(Density & density)

Binds a density field to be advected.

Parameters

- `density`: density field

VORTEX_API void Vortex::Fluid::Advection::Advect()

Performs an advection of the density field. Asynchronous operation.

VORTEX_API void Vortex::Fluid::Advection::AdvectParticleBind(Renderer::GenericBuffe

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

VORTEX_API void Vortex::Fluid::Advection::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

VORTEX_API void Vortex::Fluid::Cfl::Compute()

Compute the CFL number. Non-blocking.

VORTEX_API float Vortex::Fluid::Cfl::Get()

Returns the CFL number. Blocking.

Return cfl number

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

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

Public Functions

VORTEX_API **Circle** (*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.

VORTEX_API void Vortex::Fluid::Circle::Initialize(const Renderer::RenderState & r

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.

VORTEX_API void Vortex::Fluid::Circle::Update(const glm::mat4 & projection, const g

Update the MVP matrix of the drawable.

Parameters

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

VORTEX_API void Vortex::Fluid::Circle::Draw(Renderer::CommandEncoder & commandEnco

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** Vortex::*Fluid*::*LinearSolver*

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

Public Functions

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

Initialize the solver with a size and preconditioner.

Parameters

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

VORTEX_API void Vortex::Fluid::ConjugateGradient::Bind(Renderer::GenericBuffer & d

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

VORTEX_API void Vortex::Fluid::ConjugateGradient::BindRigidbody(float delta, Render

Bind rigidbody with the linear solver's matrix.

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

VORTEX_API void Vortex::Fluid::ConjugateGradient::Solve(Parameters & params, const

Solve iteratively solve the linear equations in data.

VORTEX_API float Vortex::Fluid::ConjugateGradient::GetError() **Return**

the max error

class Contour

#include <Boundaries.h> The *Contour* class.

class Density : **public** Vortex::*Renderer*::*RenderTexture*, **public** Vortex::*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** Vortex::*Fluid*::*Preconditioner*

#include <Diagonal.h> *Diagonal* preconditioner. Simplest of preconditioner, useful to verify if the preconditioned conjugate gradient works.

Public Functions

VORTEX_API void Vortex::Fluid::Diagonal::Bind(Renderer::GenericBuffer & d, Rendere

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** (*Renderer*::CommandEncoder &*command*)

Record the preconditioner.

Parameters

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

class DistanceField : **public** Vortex::*Renderer*::*AbstractSprite*

#include <Boundaries.h> Sprite of a distance field.

Public Functions

VORTEX_API **DistanceField** (*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

VORTEX_API void Vortex::Fluid::DistanceField::Draw(Renderer::CommandEncoder & comm

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

VORTEX_API void Vortex::Fluid::Extrapolation::Extrapolate()

Will extrapolate values from buffer into the dirichlet and neumann boundaries.

VORTEX_API void Vortex::Fluid::Extrapolation::ConstrainBind(Renderer::Texture & so

Binds a solid level set to use later and constrain the velocity against.

Parameters

- `solidPhi`: solid level set

VORTEX_API void Vortex::Fluid::Extrapolation::ConstrainVelocity()

Constrain the velocity, i.e. ensure that the velocity normal to the solid level set is 0.

class GaussSeidel : **public** Vortex::*Fluid*::*LinearSolver*, **public** Vortex::*Fluid*::*Preconditioner*

#include <GaussSeidel.h> An iterative black and red successive over relaxation linear solver.

Public Functions

VORTEX_API void Vortex::Fluid::GaussSeidel::Bind(Renderer::GenericBuffer & d, Renc

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

VORTEX_API void Vortex::Fluid::GaussSeidel::BindRigidbody(float delta, Renderer::Ge

Bind rigidbody with the linear solver's matrix.

Parameters

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

VORTEX_API void Vortex::Fluid::GaussSeidel::Solve(Parameters & params, const std::

Iterative solving of the linear equations in data.

VORTEX_API float Vortex::Fluid::GaussSeidel::GetError() **Return**

the max error

void **Record**(*Renderer*::CommandEncoder &*command*)

Record the preconditioner.

Parameters

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

void **Record**(*Renderer*::CommandEncoder &*command*, int *iterations*)

Record a determined number of iterations.

Parameters

- `commandBuffer`:
- `iterations`:

VORTEX_API void Vortex::Fluid::GaussSeidel::SetW(float w)

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

Parameters

- `w`:

VORTEX_API void Vortex::Fluid::GaussSeidel::SetPreconditionerIterations(int iterati

set number of iterations to be used when GS is a preconditioner

Parameters

- `iterations`:

class IncompletePoisson : **public** Vortex::*Fluid*::*Preconditioner*

#include <IncompletePoisson.h> Incomplete poisson preconditioner. Slightly better than a simple diagonal preconditioner.

Public Functions

VORTEX_API void Vortex::Fluid::IncompletePoisson::Bind(Renderer::GenericBuffer & c

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**(*Renderer*::CommandEncoder &*command*)

Record the preconditioner.

Parameters

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

class Jacobi : **public** Vortex::*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** (*Renderer*::CommandEncoder &*command*)

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** Vortex::*Renderer*::*RenderTexture*

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

Public Functions

VORTEX_API void Vortex::Fluid::LevelSet::Reinitialise()

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

VORTEX_API void Vortex::Fluid::LevelSet::ExtrapolateBind(Renderer::Texture & solid

Bind a solid level set, which will be used to extrapolate into this level set.

Parameters

- `solidPhi`:

VORTEX_API void Vortex::Fluid::LevelSet::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 *Vortex::Fluid::ConjugateGradient*, *Vortex::Fluid::GaussSeidel*, *Vortex::Fluid::Multigrid*

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

virtual float **GetError**() = 0

Return the max error

struct Data

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

struct DebugCopy

#include <LinearSolver.h> Copies the linear solver data in the debug linear solver data.

Public Functions

VORTEX_API void Vortex::Fluid::LinearSolver::DebugCopy::Copy()

Copies the linear solver data in the debug linear solver data.

struct DebugData

#include <LinearSolver.h> Contains the linear equations as texture, so it can easily be visualised in RenderDoc.

class Error

#include <LinearSolver.h> Calculates the max residual error of the linear system.

Public Functions

VORTEX_API void Vortex::Fluid::LinearSolver::Error::Bind(Renderer::GenericBuffe

Bind the linear system.

Parameters

- `d`: the diagonal of the matrxi

- `l`: the lower matrix
- `b`: the right hand side
- `x`: the unknowns

`VORTEX_API Error& Vortex::Fluid::LinearSolver::Error::Submit()`

Submit the error calculation.

Return this.

`VORTEX_API Error& Vortex::Fluid::LinearSolver::Error::Wait()`

Wait for error to be calculated.

Return this.

`VORTEX_API float Vortex::Fluid::LinearSolver::Error::GetError()`

Get the maximum error.

Return The error.

`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

VORTEX_API **`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`** Vortex::*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 VORTEX_API Vortex::Fluid::LocalGaussSeidel::Bind(Renderer::GenericBuffer & d,

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** (*Renderer*::CommandEncoder &*command*)

Record the preconditioner.

Parameters

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

class Multigrid : **public** Vortex::*Fluid*::*LinearSolver*, **public** Vortex::*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

VORTEX_API **Multigrid** (*Renderer*::*Device* &*device*, **const** glm::ivec2 &*size*, float *delta*, int *numSmoothingIterations* = 3, SmootherSolver *smoother* = SmootherSolver::Jacobi)

Initialize multigrid for given size and delta.

Parameters

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

VORTEX_API void Vortex::Fluid::Multigrid::Bind(Renderer::GenericBuffer & d, Render

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

VORTEX_API void Vortex::Fluid::Multigrid::BuildHierarchiesBind(Pressure & pressure

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

VORTEX_API void Vortex::Fluid::Multigrid::BuildHierarchies()

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

void **Record** (*Renderer*::CommandEncoder &*command*)

Record the preconditioner.

Parameters

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

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

VORTEX_API void Vortex::Fluid::Multigrid::Solve(Parameters & params, const std::ve
Solves the linear equations.

Parameters

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

VORTEX_API float Vortex::Fluid::Multigrid::GetError() **Return**
the max error

class ParticleCount : **public** Vortex::*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

VORTEX_API void Vortex::Fluid::ParticleCount::Scan()
Count the number of particles and update the internal data structures.

VORTEX_API int Vortex::Fluid::ParticleCount::GetTotalCount()
Calculate the total number of particles and return it.

Return

VORTEX_API Renderer::IndirectBuffer<Renderer::DispatchParams>& Vortex::Fluid::Parti
Calculate the dispatch parameters to use on the particle buffer.

Return

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

Parameters

- `levelSet`:

VORTEX_API void Vortex::Fluid::ParticleCount::Phi()
Calculate the level set from the particles.

VORTEX_API void Vortex::Fluid::ParticleCount::VelocitiesBind(Velocity & velocity,
Bind the velocities, used for advection of the particles.

Parameters

- `velocity`:
- `valid`:

VORTEX_API void Vortex::Fluid::ParticleCount::TransferToGrid()
Interpolate the velocities of the particles to the velocities field.

VORTEX_API void Vortex::Fluid::ParticleCount::TransferFromGrid()
Interpolate the velocities field in to the particles' velocity.

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

Subclassed by *Vortex::Fluid::Rectangle*

Public Functions

VORTEX_API **Polygon** (*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.

VORTEX_API void Vortex::Fluid::Polygon::Initialize(const Renderer::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.

VORTEX_API void Vortex::Fluid::Polygon::Update(const glm::mat4 & projection, const

Update the MVP matrix of the drawable.

Parameters

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

VORTEX_API void Vortex::Fluid::Polygon::Draw(Renderer::CommandEncoder & commandEnc

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 *Vortex::Fluid::Diagonal*, *Vortex::Fluid::GaussSeidel*, *Vortex::Fluid::IncompletePoisson*, *Vortex::Fluid::Jacobi*, *Vortex::Fluid::LocalGaussSeidel*, *Vortex::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** (*Renderer*::CommandEncoder &*command*) = 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

VORTEX_API void Vortex::Fluid::Pressure::BuildLinearEquation()

Build the matrix A and right hand side b.

VORTEX_API void Vortex::Fluid::Pressure::ApplyPressure()

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

class Rectangle : **public** Vortex::*Fluid*::*Polygon*

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

Public Functions

VORTEX_API **Rectangle**(*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.

VORTEX_API void Vortex::Fluid::Rectangle::Initialize(const Renderer::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.

`VORTEX_API void Vortex::Fluid::Rectangle::Update(const glm::mat4 & projection, cons`

Update the MVP matrix of the drawable.

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

`VORTEX_API void Vortex::Fluid::Rectangle::Draw(Renderer::CommandEncoder & commandB`

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 *Vortex::Fluid::ReduceJ*, *Vortex::Fluid::ReduceMax*, *Vortex::Fluid::ReduceSum*

Public Functions

`VORTEX_API Reduce::Bound Vortex::Fluid::Reduce::Bind(Renderer::GenericBuffer & inp`

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

`VORTEX_API void Vortex::Fluid::Reduce::Bound::Record(Renderer::CommandEncoder &`

Record the reduce operation.

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

`class ReduceJ` : **`public`** Vortex::*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

VORTEX_API **`ReduceJ`**(*Renderer*::*Device* &*device*, int *size*)

Initialize reduce with device and 2d size.

Parameters
- `device`:
- `size`:

class ReduceMax : **public** Vortex::*Fluid*::*Reduce*

#include <Reduce.h> *Reduce* operation on float with max of absolute.

Public Functions

VORTEX_API **ReduceMax** (*Renderer*::*Device* &*device*, int *size*)

Initialize reduce with device and 2d size.

Parameters

- device:
- size:

class ReduceSum : **public** Vortex::*Fluid*::*Reduce*

#include <Reduce.h> *Reduce* operation on float with addition.

Public Functions

VORTEX_API **ReduceSum** (*Renderer*::*Device* &*device*, int *size*)

Initialize reduce with device and 2d size.

Parameters

- device:
- size:

class RigidBody : **public** Vortex::*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 VORTEX_API void Vortex::Fluid::RigidBody::ApplyForces()

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

virtual VORTEX_API void Vortex::Fluid::RigidBody::ApplyVelocities()

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

VORTEX_API void Vortex::Fluid::RigidBody::SetMassData(float mass, float inertia)

Sets the mass and inertia of the rigidbody.

Parameters

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

VORTEX_API void Vortex::Fluid::RigidBody::SetVelocities(const glm::vec2 & velocity,

sets the velocities and angular velocities of the body

Parameters

- velocity:
- angularVelocity:

VORTEX_API void Vortex::Fluid::RigidBody::UpdatePosition()

Upload the transform matrix to the GPU.

VORTEX_API void Vortex::Fluid::RigidBody::RenderPhi()

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

VORTEX_API void Vortex::Fluid::RigidBody::BindPhi(Renderer::RenderTexture & phi)

Bind the rendertexture where this rigidbodies shape will be rendered.

Parameters

- `phi`: render texture of the world

VORTEX_API void Vortex::Fluid::RigidBody::BindDiv(Renderer::GenericBuffer & div, R

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

VORTEX_API void Vortex::Fluid::RigidBody::BindVelocityConstrain(Fluid::Velocity &

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

Parameters

- `velocity`:

VORTEX_API void Vortex::Fluid::RigidBody::BindForce(Renderer::GenericBuffer & d, R

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

Parameters

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

VORTEX_API void Vortex::Fluid::RigidBody::BindPressure(float delta, Renderer::Gener

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

Parameters

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

VORTEX_API void Vortex::Fluid::RigidBody::Div()

Apply the body’s velocities to the linear equations matrix A and right hand side b.

VORTEX_API void Vortex::Fluid::RigidBody::Force()

Apply the pressure to body, updating its forces.

VORTEX_API void Vortex::Fluid::RigidBody::Pressure()

Reduce the force for pressure update.

VORTEX_API void Vortex::Fluid::RigidBody::VelocityConstrain()

Constrain the velocities field based on the body’s velocity.

VORTEX_API Velocity Vortex::Fluid::RigidBody::GetForces()

Download the forces from the GPU and return them.

Return

VORTEX_API Type Vortex::Fluid::RigidBody::GetType()

Type of this body.

Return

VORTEX_API void Vortex::Fluid::RigidBody::SetType(Type type)

Set the type of the body.

Parameters

- `type`:

VORTEX_API Renderer::RenderTexture& Vortex::Fluid::RigidBody::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** Vortex::*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

VORTEX_API void Vortex::Fluid::SmokeWorld::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

VORTEX_API **Transfer**(*Renderer*::*Device* &*device*)

Initialize prolongate and restrict compute pipelines.

Parameters

- `device`:

VORTEX_API void Vortex::Fluid::Transfer::ProlongateBind(std::size_t level, const gl

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`

VORTEX_API void Vortex::Fluid::Transfer::RestrictBind(std::size_t level, const glm:

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`

`VORTEX_API void Vortex::Fluid::Transfer::Prolongate(Renderer::CommandEncoder & com`

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.

`VORTEX_API void Vortex::Fluid::Transfer::Restrict(Renderer::CommandEncoder & comma`

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`** Vortex::*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 Types

`enum InterpolationMode`

Velocity interpolation when querying in the shader with non-integer locations.

Values:

`Linear` = 0

`Cubic` = 1

Public Functions

`VORTEX_API Renderer::Texture& Vortex::Fluid::Velocity::Output()`

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

Return

`VORTEX_API Renderer::Texture& Vortex::Fluid::Velocity::D()`

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

Return

`VORTEX_API void Vortex::Fluid::Velocity::CopyBack(Renderer::CommandEncoder & comma`

Copy the output field to the main field.

Parameters

- `commandBuffer`:

`VORTEX_API void Vortex::Fluid::Velocity::Clear(Renderer::CommandEncoder & command)`

Clear the velocity field.

Parameters

- `commandBuffer`:

`VORTEX_API void Vortex::Fluid::Velocity::SaveCopy()`

Copy to the difference field.

`VORTEX_API void Vortex::Fluid::Velocity::VelocityDiff()`

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

`class WaterWorld` : **`public`** Vortex::*Fluid*::*World*

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

Public Functions

`VORTEX_API Renderer::RenderCommand Vortex::Fluid::WaterWorld::RecordParticleCount(F`

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

`VORTEX_API void Vortex::Fluid::WaterWorld::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 *Vortex::Fluid::SmokeWorld*, *Vortex::Fluid::WaterWorld*

Public Functions

`World`(*Renderer*::*Device* &*device*, **`const`** glm::ivec2 &*size*, float *dt*, int *numSubSteps* = 1, *Velocity*::InterpolationMode *interpolationMode* = *Velocity*::InterpolationMode::Linear)

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.

`VORTEX_API void Vortex::Fluid::World::Step(LinearSolver::Parameters & params)`

Perform one step of the simulation.

`VORTEX_API Renderer::RenderCommand Vortex::Fluid::World::RecordVelocity(Renderer::F`

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

VORTEX_API void Vortex::Fluid::World::SubmitVelocity(Renderer::RenderCommand & ren

submit the render command created with *RecordVelocity*

Parameters

- `renderCommand`: the render command

VORTEX_API Renderer::RenderCommand Vortex::Fluid::World::RecordLiquidPhi(Renderer::

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

VORTEX_API Renderer::RenderCommand Vortex::Fluid::World::RecordStaticSolidPhi(Rende

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

VORTEX_API std::shared_ptr<DistanceField> Vortex::Fluid::World::MakeLiquidDistanceF

Create sprite that can be rendered to visualize the liquid level set.

Return a sprite

VORTEX_API std::shared_ptr<DistanceField> Vortex::Fluid::World::MakeSolidDistanceFi

Create sprite that can be rendered to visualize the solid level set.

Return a sprite

VORTEX_API void Vortex::Fluid::World::AddRigidbody(RigidBody & rigidbody)

Add a rigibody to the solver.

Parameters

- `rigidbody`:

VORTEX_API void Vortex::Fluid::World::RemoveRigidBody(RigidBody & rigidbody)

Remove a rigidbody from the solver.

Parameters

- `rigidbody`:

VORTEX_API void Vortex::Fluid::World::AttachRigidBodySolver(RigidBodySolver & rigi

Attach a rigidbody solver, e.g. box2d.

Parameters

- `rigidbodySolver`:

VORTEX_API float Vortex::Fluid::World::GetCFL()

Calculate the CFL number, i.e. the width divided by the max velocity.

Return CFL number

VORTEX_API Renderer::Texture& Vortex::Fluid::World::GetVelocity()

Get the velocity, can be used to display it.

Return velocity field reference

Index