GIVR Documentation

Release 0.0.1

Lakin Wecker

Jan 21, 2020

QUICK START

givr is a GPLv3 licensed, type-safe API for rendering geometry. It was primarily developed by Lakin Wecker, Jeremy Hart, Andrew Owens, Kathleen Ang with guidance from Dr. Przemyslaw Prusinkiewicz. It was developed for use in CPSC 587 - Computer Animation at the University of Calgary.

Computer graphics programs are incredibly rewarding. Not only do you get immediate feedback about your program when you run it, the results are enjoyable to watch and can be shown to your friends and family. However, modern Computer Graphics programming environments are complicated. They are designed to allow for maximum flexibility and performance and leave you to do much of the verification yourself. It is very easy to make a program that compiles, runs and produces no output on the screen.

givr provides an API for getting basic geometry onto the screen with few lines of code and with a type-safe API that prevents many common errors. It is written using C++-17 and provides syntax that allows you to quickly specify geometry details, style parameters and the camera/projection (view) such that the geometry appears on screen.

It makes use of C++ templates and type checking to ensure that only compatible instances of each are used together and that all required parameters are specified.

givr requires that you provide an OpenGL context and allows you to use any windowing and user input library. It also requires that you provide the main loop for your program.

This documentation is split into three parts. The first part is a general introduction to graphics programming and will cover how graphics programs are often organized and which concepts are necessary. The second part is a user manual, which guides you through downloading, building and running your first givr based program. The third is a reference manual that covers all of the API and their parameters.

CHAPTER

ONE

QUICK START

You must have a compiler which can compile C++17. Thankfully, most recent compilers release in the past couple of years (written late 2019) will do a sufficient job for givr.

1.1 Building the Examples

There are some simple examples you can use to get started with givr. Included within them are all necessary dependencies to compile except for a compiler and CMake.

You will need a compiler capable of compiling C++17. We have tested these projects on various recent distributions and it compiles and runs.

Download a recent version of the examples. They are included as part of the givr distribution, which can be downloaded here: https://github.com/giv-lab/givr/archive/v0.0.11.zip Unzip this file somewhere. Then continue to your OS specific instructions.

You can also use git to grab a copy, the gitlab repository is here: https://github.com/giv-lab/givr/tree/v0.0.11

1.1.1 Windows

On windows, you will need Visual Studio 2017. You may also need to update it after installation so that your compiler is up to date (launch Visual Studio and click on the flag in the top corner). You will also need to download and install cmake: https://cmake.org/download/

Create a `build` directory which will contain all of the build files. You can create this anywhere, but I most often create it next to or within the directory that was created above.

Now run the cmake-gui command from your start menu. It requires that you specify two directories. The first is the directory which contains the CMakeLists.txt you wish to build. The second is the `build` directory that you created.

Once these have been selected, press the configure button. It will run for a few seconds, produce some output in the bottom frame and then show you some configuration entries that you can change if you would like. You do not need to change any of these, despite the fact that they are coloured red. Next, press the generate button. Finally, press the Open Project button.

Note that by default the project will not have a default startup file set. You can choose one of the example programs as your default and then build and run it.

If you want more information on how to use cmake-gui on windows they have instructions here: https://cmake.org/runningcmake/

Windows Video

1.1.2 Linux

In linux you will need to install cmake. All modern distributions come with a way to install it that is pretty simple or straightforward. In Ubuntu you can use:

```
sudo apt-get install cmake build-essential
```

In arch linux you can use:

```
pacman -Sy cmake
```

Once cmake is installed, navigate to the directory which contains the *CMakeLists.txt* file and run these commands:

```
cmake -H. -Bbuild -DCMAKE_BUILD_TYPE=Release
cmake --build build
```

This will create a *build* directory and build the project within it. You can then run the examples afterwards with:

```
./build/sph
./build/pbd
```

1.2 Using the example project

There is a sample project you can use as a basis for your project. It is available at the link below and comes with a cmake file for building your project. The build instructions follow those given for the examples above.

Download the project here:

- example-project

1.3 Including givr in an existing project.

We have also provided a simple way to get givr up and running for your own project. If you already have a project and want to integrate it directly, then follow these steps

1.3.1 Step 1: Download givr.h

givr is distributed as two files, `givr.h` which you include in your programs and a `givr.cpp` which you must compile and link with your programs.

They can be downloaded here:

- givr.h
- givr.cpp

1.3.2 Step 2: Obtain a windowing library

If you already have a windowing library setup, you may skip this step. If not, then read on for a brief overview of how to setup windowing/opengl in your program.

You can use any windowing library with givr. However, for this documentation, we will use `glfw3`. There are many ways to obtain and install `glfw3`. We recommend (vcpkg) <https://github.com/Microsoft/vcpkg>. You will also need a method for initalizing the OpenGL libraries. We are using `glad` in the sample project and examples, which may also be installed via `vcpkg`.

Finally, we will also use a set of helpers that wraps `glad` and `glfw3` into a simpler API. This library is called `givio` and is available here:

- io.h

1.3.3 Step 3: Your main program

You will need a main.cpp to run your program. You can write your own or start from this simple example of a triangle:

- triangle.cpp

CHAPTER

TWO

INTRODUCTION

Your job as a 3D graphics programmer is to put pixels on the screen. Figuring out how to do that with modern computer graphics pipelines like OpenGL or Vulkan is a daunting and complicated task. Given the plethora of concepts, data structures, api calls and opaque state that you must deal with, the complexity can be bewildering. However, the resulting programs are some of the most fun and rewarding experiences that you can have as a programmer.

When you first start out, you may not realize how many concepts are necessary to get a fully functional graphics program running. This introduction will cover all of this and provide you with some example libraries and code that you can use to get started.

A graphics program will typically need to do all of the following:

- Initialize a window using the user interface libraries for the operating system
- Initialize communication with the graphics card drivers
- Setup OpenGL (or Vulkan or Direct3D) appropriately in preparation for rendering
- Handle keyboard and mouse input from the user
- Load the resources you will use for rendering (models, textures, etc).
- Upload these resources to the GPU
- Provide programmable instructions to the GPU on how to render those resources
- Integrate and use a system for displaying graphical widgets like dropdowns and buttons
- Math related utilities for vector, matrix and opengl related operations.

Thankfully you will rarely (if ever?) need to deal with each of these tasks directly. You will almost always use a set of libraries which helps you deal with them and may even provide a cross-platform API that you may use. There are many options for libraries that deal with these requirements and I cannot list them all, however some of the ones I have used and can recommend are:

1. Simple Direct Media Library (SDL): a very popular cross platform library for dealing with user input, window creation, OpenGL context setup image loading, text rendering, sound and many other items.
2. Graphics Library FrameWork (GLFW): another popular cross platform library for dealing with user input, window creation and OpenGL context setup.
3. OpenGL Mathematics (GLM): A header only C++ mathematics library for graphics software which is based on the OpenGL Shading Language Specifications.
4. Eigen: Eigen is a C++ template library for linear algebra: matrices, vectors, numerical solvers, and related algorithms.
5. There are many. For a more comprehensive list see the Khronos Related Toolkits and APIS page.

For windowing and OpenGL context management we will be using the excellent givio library written by Andrew Owens. It wraps GLFW into a simple to use, C++-17 interface which greatly simplifies its setup and usage.

For mathematics, we will be using the GLM library as it closely follows the GLSL specification which reduces cognitive overhead when you switch between writing GLSL and C++.

The following code is an example of a program uses `givr`, `givio` and `glm`. The rest of this introduction will slowly build up to this final program, and each line will be explained.

```
#include "givr.h"
#include <glm/gtc/matrix_transform.hpp>

#include "io.h"

using namespace glm;
using namespace givr;
using namespace givr::camera;
using namespace givr::geometry;
using namespace givr::style;

int main(void)
{
    io::GLFWContext windows;
    auto window = windows.create(io::Window::dimensions{640, 480}, "Simple example
↪");
    window.enableVsync(true);

    window.keyboardCommands()
      | io::Key(GLFW_KEY_ESCAPE,
          [&](auto const &/*event*/) { window.shouldClose(); });

    auto view = View(TurnTable(), Perspective());

    auto triangle = createRenderable(
        Triangle(Point1(0.0, 1., 0.), Point2(-1., -1., 0.), Point3(1., -1., 0.)),
        Phong(Colour(1., 1., 0.1529), LightPosition(2., 2., 15.))
    );

    glClearColor(1.f, 1.f, 1.f, 1.f);
    float u = 0.;
    window.run([&](float frameTime) {
        view.projection.updateAspectRatio(window.width(), window.height());
        mat4f m{1.f};
        u += frameTime;
        auto angle = 365.f*sin(u*.01f);
        m = rotate(m, angle, vec3f{1.0, 1.0, 0.0});
        auto size = cos(u*0.1f);
        m = scale(m, 15.f*vec3f{size});
        draw(triangle, view, m);
    });
    exit(EXIT_SUCCESS);
}
```

CHAPTER

THREE

WINDOWING AND INPUT WITH *GIVIO*

The first thing your graphics program must do is to create a window and OpenGL context. Most libraries which deal with window creation will also deal with OpenGL context creation in some fashion. They will almost always deal with user input as well.

The `givio` library does all three for us, and does it with very few lines of code. The following is a complete sample of using `givio` to create a window (and OpenGL context), respond to user input (exit when the escape key is pressed) and then show the window until the program is exited.

```
1   #include "io.h"
2
3   int main(void)
4   {
5       io::GLFWContext windows;
6       auto window = windows.create(io::Window::dimensions{640, 480}, "Simple example
    ↪");
7       window.enableVsync(true);
8
9       window.keyboardCommands()
10        | io::Key(GLFW_KEY_ESCAPE,
11            [&](auto const &/*event*/) { window.shouldClose(); });
12
13      window.run([&](float frameTime) {
14          // Do Nothing
15      });
16      exit(EXIT_SUCCESS);
17  }
```

Lines, 3, 4, 16, and 17 (shown below) are all standard C++ code for a program. We won't describe them further.

```
int main(void)
{
    exit(EXIT_SUCCESS);
}
```

Line 1 (shown below) includes the `givio` header and makes it available to us to use.

```
#include "io.h"
```

Line 5 and 6 (shown below) create the context and then a window which is sized to be 640x480 pixels large and has a title of "Simple example".

```
io::GLFWContext windows;
auto window = windows.create(io::Window::dimensions{640, 480}, "Simple example");
```

Line 7 (shown below) is an optional line which enables a commonly used technique called v-sync. This technique ensures that whenever the screen is rendered, it's rendered just before your monitor updates the image. Doing so avoids the 'tearing' artifact that may otherwise be present. However, it does limit your program to run only as fast as your monitor's refresh rate.

```
window.enableVsync(true);
```

Line 9-11 (shown below) shows how you can respond to keyboard input with `givio`. In this case, we responsd to the escape key by asking the window to close.

```
window.keyboardCommands()
  | io::Key(GLFW_KEY_ESCAPE,
      [&](auto const &/*event*/) { window.shouldClose(); });
```

Line 13-15 (shown below) is how you ask `givio` to run a main loop. You might notice that we previously said that `givr` does not provide a main loop. This is true and intentional so that you can fully control how your program runs. However, `givio` does provide a main loop as a convenience for those of you that do not want to control every last aspect.

```
window.run([&](float frameTime) {
    // Do Nothing
});
exit(EXIT_SUCCESS);
```

In summary, `givio` is a library that provides a succinct way to create a window, OpenGL context and respond to user input. It makes a great partner library for `givr`.

CHAPTER

FOUR

RENDERING WITH *GIVR*

Here is a very simple program that renders a rotating triangle. We will use this example to get a basic idea of what givr does and how you use it. The example comes from the givr-examples repository. If you successfully built the examples using the instructions above, then you can open the examples/triangle.cpp file in your code editor and follow along from there.

This example also uses the *io.h* library that Andrew Owens created for managing GLFW, and the turntable controls from the givr-examples. The lines which are highlighted in yellow are example of the givr API. We will talk about each of these lines in the following section. The rest of the lines come from C++, glm, *io.h* or the turntable controls.

```
1  //--------------------------------------------------------------------------
2  // A simple example showing how to use the triangle geometry
3  //--------------------------------------------------------------------------
4  #include "givr.h"
5  #include <glm/gtc/matrix_transform.hpp>
6
7  #include "io.h"
8  #include "turntable_controls.h"
9
10 using namespace glm;
11 using namespace givr;
12 using namespace givr::camera;
13 using namespace givr::geometry;
14 using namespace givr::style;
15
16 int main(void)
17 {
18     // Set up your windowing system / OpenGL context
19     io::GLFWContext windows;
20     auto window = windows.create(io::Window::dimensions{640, 480}, "Simple example
   ↪");
21
22     auto view = View(TurnTable(), Perspective());
23     TurnTableControls controls(window, view.camera);
24
25     auto triangle = createRenderable(
26         Triangle(Point1(0.0, 1., 0.), Point2(-1., -1., 0.), Point3(1., -1., 0.)),
27         Phong(Colour(1., 1., 0.1529), LightPosition(2., 2., 15.))
28     );
29
30     glClearColor(1.f, 1.f, 1.f, 1.f);
31     float u = 0.;
32     window.run([&](float frameTime) {
33         view.projection.updateAspectRatio(window.width(), window.height());
34         mat4f m{1.f};
```

(continues on next page)

(continued from previous page)

```
35            u += frameTime;
36            auto angle = 365.f*sin(u*.01f);
37            m = rotate(m, angle, vec3f{1.0, 1.0, 0.0});
38            auto size = cos(u*0.1f);
39            m = scale(m, 15.f*vec3f{size});
40            draw(triangle, view, m);
41        });
42        exit(EXIT_SUCCESS);
43    }
```

4.1 givr program anatomy

There are 8 things that need to be in place for givr to render things to the screen:

1. Include givr.h
2. Using Namespace (optional)
3. Instantiate camera/view
4. Instantiate your geometry
5. Instantiate your style
6. Create the renderable
7. (Optional) Add instances
8. Draw

4.1.1 1. Include givr.h

Just like all C++ libraries, you must include it before you use it:

```
#include <givr.h>
```

The triangle example also includes glm (for doing math), io.h (for handling windowing; it essentially wraps GLFW), and turntable_controls.h (for interacting with the scene, e.g. rotating and zooming in/out).

4.1.2 2. Using Namespace (optional)

givr uses namespaces to organize its code. In most of the examples we make use of using namespace directives to shorten the amount of code we have to type. How much you use this is up to you:

```
using namespace givr;
using namespace givr::camera;
using namespace givr::geometry;
using namespace givr::style;
```

4.1.3 3. Instantiate Camera/View

givr comes with a builtin camera and projection class:

```
auto view = View(TurnTable(), Perspective());
```

When your window changes size, you will want to inform the projection class of the change in aspect ratio. To do this you use the `view.project.updateAspectRatio` method:

```
view.projection.updateAspectRatio(width, window);
```

You will need to get the width and height values from somewhere. If you are using the *io.h* library, then you can ask for them directly from the window class that you instantiated:

```
io::GLFWContext windows;
auto window = windows.create(io::Window::dimensions{640,480}, "Simple example");
...
view.projection.updateAspectRatio(window.width(), window.height());
```

4.1.4 4. Instantiate Geometry

givr comes with a number of different types of geometry, e.g. lines, triangles, spheres, a Mesh loaded from an OBJ file, and custom geometry.

Note that when you instantiate the geometry object, you are not actually building the geometry. It isn't until you create the renderable that the geometry is created. In our triangle example code, we've basically rolled steps 4-6 (instantiating geometry, instantiating style, and creating the renderable) into one call:

```
auto triangle = createRenderable(
    Triangle(Point1(0.0, 1., 0.), Point2(-1., -1., 0.), Point3(1., -1., 0.)),
    Phong(Colour(1., 1., 0.1529), LightPosition(2., 2., 15.))
);
```

To instantiate the triangle only:

```
auto triangle = Triangle(Point1(0.0, 1., 0.), Point2(-1., -1., 0.), Point3(1., -1., 0.
↪));
```

See *Geometry* for more details on all of the types of geometry that are supported.

4.1.5 5. Instantiate Style

givr comes with two different styles: a smooth shaded phong style and a line style for rendering lines. We saw above how the style instantiation was included in the createRenderable call, but we could also instantiate it separately. For example, Phong style instantiation could look like:

```
auto phongStyle = Phong(
     Colour(1.0, 1.0, 0.1529),
     LightPosition(2.0, 2.0, 15.0)
 );
```

See *Styles* for more details on all of the types of styles that are supported.

4.1.6 6. Create the renderable

There are two types of renderables in givr: instanced and non-instanced. Instanced geometry is used when you need to render many of the same object in a scene where the only difference is the position and orientation of those objects

(for example, you could be drawing many balls falling into a bowl – see the example *pbd*). Non-instanced geometry is slightly easier to use, but requires a draw call for each instance.

We have already seen an example of creating the non-instanced renderable:

```
auto triangle = createRenderable(
    Triangle(Point1(0.0, 1., 0.), Point2(-1., -1., 0.), Point3(1., -1., 0.)),
    Phong(Colour(1., 1., 0.1529), LightPosition(2., 2., 15.))
);
```

An example of creating the instanced renderable:

```
auto instancedSpheres = createInstancedRenderable(Sphere(), phongStyle);
```

4.1.7 7. (Optional) Add instances

If you are using the instanced renderable, then you must add individual instances using the *addInstance* function. It takes the renderable as the first parameter and a 4x4 model matrix as the second renderable. (For a working example, refer to *pbd*.)

You can use glm matrix transform functions to instantiate the matrix: https://glm.g-truc.net/0.9.2/api/a00245.html

Adding instances looks approximately like this:

```
// Use GLM to translate to a specific location.
mat4f m = translate(mat4f{1.f}, vec3f{0., 5.0, 0.});
addInstance(instancedSpheres, m);
```

4.1.8 8. Draw

When you are ready to draw, simply call the draw command. Please note that givr does not clear the screen for you. You should remember to clear the screen yourself using something like:

```
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

The draw calls for both instanced and non-instanced renderables are nearly identical with one minor difference. The following syntax works with both:

```
draw(instancedSpheres, view);
```

In this version you pass in only your renderable and the view you created with the associated camera/projection objects. If your renderable is an instanced renderable it will draw as many instances as you set up when you called *addInstance*. If your renderable is not an instanced renderable then it will draw a single instance with the identity matrix as the model transform matrix. This usually places the geometry at the origin.

If you have a non-instanced renderable, there is an alternative form of the draw command that you can use to place the object somewhere else. You can pass a third parameter which is the model transformation matrix for this particular draw call, which is what we see in our triangle example:

```
mat4f m{1.f};
// Update model matrix, m:
u += frameTime;
auto angle = 365.f*sin(u*.01f);
m = rotate(m, angle, vec3f{1.0, 1.0, 0.0});
auto size = cos(u*0.1f);
```

(continues on next page)

(continued from previous page)

```
m = scale(m, 15.f*vec3f{size});
// Alternate call to draw:
draw(triangle, view, m);
```

Once again you can use the glm matrix transformation functions to construct the appropriate matrix.

CHAPTER

FIVE

CAMERA AND PROJECTION

The camera and projection system within givr is still undergoing quite work to design it to be more friendly and easy to use. As with all of givr, feedback is welcome.

There are three components involved in this system: the view context, the camera and the projection.

5.1 View Context

The view context is provided by givr and is a simple templated struct which holds a reference to a camera and projection class.

In the example code below and in the examples provided throughout the documentation, the type for the camera is `givr::camera::TurnTable` and the type for the projection is `givr::camera::PerspectiveView`. The instance of the camera is available via the `view.camera` attribute and the instance of the projection is available via the `view.projection` attribute. Both types must be able to be instantiated with no parameters.

5.1.1 Example

```
auto view = View(TurnTable(), Perspective());
```

5.2 Camera

The only camera type built into givr at the moment is a TurnTable camera, which provides some simple TurnTable type controls. You can rotate around two axes and zoom in and out.

If you would like to provide your own camera, you may do so. In order to do so you must do four things:

1. Create a custom type (struct/class) that represents your camera.
2. Define a *mat4f getViewMatrix(MyCameraT const &camera)* function which within that class generates the view matrix from the camera instance.
3. Define a *mat4f getViewPosition(MyCameraT const &camera)* function which within that class generates the view position for given camera instance.
4. Specify this type of camera for your view context

The actual struct/class that you define may have as many attributes and methods as needed. None of them will be used by givr directly. Whenever the style needs the view position and view matrix, it will call the appropriate method defined above.

5.2.1 Example

```
struct MyCamera {
    static mat4f getViewMatrix(MyCamera const &c) {
        auto up = ...;
        auto pos =  ...;
        auto lookAtPoint =  ...;
        return lookAt(pos, up, lookAtPoint);
    }

    static vec3f getViewPosition(MyCamera const &c) {
        auto pos = ...;
        return pos;
    }
};
...
givr::ViewContext<MyCamera, givr::PerspectiveView> view;
```

5.3 Projection

Similar to the Camera class above, the projection class can also be customized. You need to do 3 things for the projection class:

1. Create a custom type (struct/class) that represents your projection.
2. Define a *mat4f getProjectionMatrix(MyProjectionT const &camera)* within that class function which generates the view matrix from the projection instance.
3. Specify this type of projection for your view context.

CHAPTER

SIX

GEOMETRY

6.1 Introduction

The following documentation gives you a terse, but reasonably complete description of the types of geometry that you can instantiate within givr.

Note: We use some Template Metaprogramming techniques to provide this API. These techniques are used so that your code will not compile when the you have provided incomplete data or incorrect types. It also allows you to provide the parameters to each function call in whatever order you choose and enforces a style that promotes readability. The name of the struct which contains the geometry has a different name from the function which you use to instantiate it. As such we make liberal use of the keyword `auto` to simplify the examples.

If you are creating a simple demo in a single file and using global variables to hold all of your data/styles/geometry then you can use the example code below in the same format.

If you intend to organize your code into classes and need to know the exact types that are used in order to declare member variables or function parameters that take these as types, then read the *Actual Types* section at the end.

6.2 Types of Geometry

givr provides the following types of geometry:

6.3 Triangle

The Triangle geometry is used to create a single triangle.

6.3.1 Parameters

It has three parameters, one for each point of the triangle:

- `Point1(float, float, float)`: The position of vertex 1
  - **Required**
- `Point2(float, float, float)`: The position of vertex 2
  - **Required**
- `Point3(float, float, float)`: The position of vertex3
  - **Required**

6.3.2 Data

The triangle geometry provides the following data to the style:

- vertices
- normal
  - **NOTE:** The normal is calculated as if the winding order is clockwise.

6.3.3 Example

```
auto triangle = Triangle(
    Point1(0.f, 1.f, 0.f),
    Point2(-1.f, -1.f, 0.f),
    Point3(1.f, -1.f, 0.f)
);
```

6.4 Line

The line geometry is for generating a single line segment.

6.4.1 Parameters

It has two parameters, one for each point of the triangle:

- `Point1(float, float, float)`: The position of vertex 1
  - **Required**
- `Point2(float, float, float)`: The position of vertex 2
  - **Required**

6.4.2 Data

The line geometry provides this data to the style:

- vertices

6.4.3 Example

```
auto line = Line(
    Point1(-15.0, -15.0, 0.0),
    Point2(15.0, 15.0, 0.0)
);
```

6.5 MultiLine

A MultiLine is simply a series of line segments which may or may not be connected. It is analogous to the GL_LINES rendering type.

6.5.1 Parameters

It takes an arbitrary number of lines as its parameters.

You can also add line segments using the following API:

```
auto l = MultiLine();
l.push_back(Line(Point1(-20.f, -20.f, 0.f), Point2(-20.f, -10.f, 0.f)));
```

6.5.2 Data

The MultiLine geometry provides this data to the style:

- vertices

6.5.3 Example

```
auto multiLine = MultiLine(
    Line(Point1(-20.f, -20.f, 0.f), Point2(-20.f, -10.f, 0.f)),
    Line(Point1(-10.f, -10.f, 0.f), Point2(-10.f, 0.f, 0.f)),
    Line(Point1(0.f, 0.f, 0.f), Point2(0.f, 10.f, 0.f)),
    Line(Point1(10.f, 10.f, 0.f), Point2(10.f, 20.f, 0.f))
);
```

6.6 Polyline

A polyline is composed of a series of points where each line segment connects the current point with the previous point. It is valid for any number of points greater than 1.

6.6.1 Parameters

It is a templated class, which takes givr::PrimitiveType as the template parameter. This template parameter may be set to one of two values:

- `givr::PrimitiveType::LINE_LOOP`
- `givr::PrimitiveType::LINE_STRIP`

If you use `PrimitiveType::LINE_LOOP`, the final point will be connected by a line segment with the first point. If you use `PrimitiveType::LINE_STRIP` then it will not be. This parameter is a template parameter so that we can do compile time checking to ensure it is set to the right value.

The class takes a list of points as parameters.

6.6.2 Data

The PolyLine geometry provides this data to the style:

- vertices

6.6.3 Example

```
auto polyline = PolyLine<PrimitiveType::LINE_LOOP>(
    Point(-10.f, -10.f, 0.f),
    Point(10.f, -10.f, 0.f),
    Point(10.f, 10.f, 0.f),
    Point(-10.f, 10.f, 0.f),
    Point(-10.f, -10.f, 0.)
);
```

6.7 Sphere

The sphere geometry is used to generate a set of triangles which approximate a sphere. By default the sphere is a unit sphere, centred around the origin. You can change where its `Centroid` and its `Radius` by providing them when you construct it, or you can use a model matrix to place it in the correct position and scale it appropriately.

6.7.1 Parameters

It has four parameters:

- `Centroid(float, float, float)`: The point around which the sphere is centred.
  - *Default*: `Centroid(0.f, 0.f, 0.f)`
- `Radius(float)`: The radius of the sphere.
  - *Default*: `Radius(1.f)`
- `AzimuthPoints(int)`: The number of azimuthal sample points to use.
  - *Default*: `AzimuthPoints(20)`
- `AltitudePoints(int)`: The number of altitude sample points to use.
  - *Default*: `AltitudePoints(20)`

6.7.2 Data

The sphere produces:

- vertices
- normals
- indices
- uvs

Note: uv coordinates are not currently used by any styles.

6.7.3 Example

Typically you will just use the sphere as is and scale it when you draw it:

```
auto instancedSpheres = createInstancedRenderable(Sphere(), phongStyle);

mat4f m = translate(mat4f{1.f}, vec3f{0., 5.0, 0.});
addInstance(instancedSpheres, m);
draw(instancedSpheres, view);
```

Alternatively, you can change its parameters directly when creating it:

```
auto spheres = createRenderable(
    Sphere(
        Centroid(1.0f, -10.f, 0.f),
        Radius(5.f),
        AzimuthPoints(5),
        AltitudePoints(5)
    ),
    phongStyle
);

draw(spheres, view);
```

6.8 Cylinder

The Cylinder geometry allows you to place a cylinder that connects two points. It's often used in place of `GL_LINES` as it is actually a 3D object, while `GL_LINES` are not.

Note: The current implementation is an open-faced cylinder.

6.8.1 Parameters

It has four parameters:

- `Point1(float, float, float)`: the first end point of the cylinder centred.
  - *Default*: `Point1(0.f, 0.5f, 0.f)`
- `Point2(float, float, float)`: the first end point of the cylinder centred.
  - *Default*: `Point1(0.f, -0.5f, 0.f)`
- `Radius(float)`: The radius of the cylinder.
  - *Default*: `Radius(1.f)`
- `AzimuthPoints(int)`: The number of azimuthal sample points to use.
  - *Default*: `AzimuthPoints(20)`

6.8.2 Data

It generates this data for the style to use:

- vertices
- normals
- indices

6.8.3 Example

```
auto cylinder = Cylinder(
    Point1(-15.0, 15.0, 0.f),
    Point2(-15.0, -15.0, 0.f)
);
```

6.9 Mesh

The Mesh geometry allows you to load arbitrary meshes from .obj files and then render them.

6.9.1 Parameters

It has a single parameter, which is the filename of the .obj. Note that it attempts to load the filename you give it directly, without modification. This means that it is your responsibility to ensure that the path will work when your executable is run. If you use relative paths, you will need to ensure that your application is always run in the same directory. If you use absolute paths then you will need to ensure there is a way to easily change that when you move the program between machines:

- `Filename(std::string)`: The filename to load
  - **Required**

6.9.2 Data

The Mesh object will produce the following data for the style to use:

- vertices
- normals
- indices
- uvs

6.9.3 Example

```
auto palmTree = Mesh(Filename("./models/Palm_Tree.obj"));
```

6.10 Triangle Soup

This is the first option for defining your own custom geometry. It's slightly easier to use, but also slightly less efficient.

Triangle soup is an affectionate name for large set of triangles representing an object, but no implicit connectivity or topology. This geometry type is like the `CustomGeometry` (described below) in that it allows you to easily build new shapes surfaces or other items, but it provides a slightly easier to use interface to do so.

NOTE: This type of geometry produces normals for each triangle, and assigns that normal to each vertex of that triangle. In addition, each vertex of the triangle is explicitly represented in the vertices array regardless of whether other triangles share the same vertex. The result of this is that they shading will not be smooth across the edges of triangles. If you want custom geometry with smooth shading, you will need to use givr::CustomGeometry (see below).

6.10.1 Parameters

It takes a list of triangles as its parameters.

You can also add triangles using the following API:

```
auto ts = TriangleSoup();
ts.push_back(
    Triangle(
        Point1(-20.f, -20.f, 0.f),
        Point2(-10.f, -10.f, 0.f),
        Point3(-20.f, 0.f, 0.f)
    )
);
```

6.10.2 Data

The triangle geometry these pieces of data which are made available to the style:

- vertices
- normals

6.10.3 Example

```
auto ts = TriangleSoup(
    Triangle(
        Point1(-20.f, -20.f, 0.f),
        Point2(-10.f, -10.f, 0.f),
        Point3(-20.f, 0.f, 0.f)
    ),
    Triangle(
        Point1(-40.f, -40.f, 0.f),
        Point2(-20.f, -20.f, 0.f),
        Point3(-40.f, -10.f, 0.f)
    )
);
```

Or more likely you will loop over the elements in your animation/simulation and turn them into a series of triangles:

```
auto ts = triangleSoup();
// Loop over all objects in your simulation/animation
for(int i = 0; i < my_simulation.objects.size(); ++i) {
   // Get a reference to the object
   object const &o = my_simulation.objects[i];

   // Turn that object into a Triangle (or triangles!)
   TriangleSoup t{o.get_point1(), o.get_point2(), o.get_point3()};

   // Add that triangle to the triangle soup
   ts.push_back(t);
}
```

As a specific example, here is how I generated the triangles for the sides of my jelly cube for the mass springs assignment. I stored my particle masses in a 1D vector, and then I painstakingly did all of the index math to generate triangles. It wasn't fun, I'm sure there are better ways:

```
auto jellyGeometry = TriangleSoup();

void updateJellyGeometry() {
    // This gets called for every frame, so it's not hyper efficient, but
    // reasonable for 60ish fps
    jellyGeometry.triangles.clear();

    auto pos = [&](std::size_t i, std::size_t j, std::size_t k) {
        return jelly.particles[(i*(resolution*resolution)) + (j*resolution) + k].
↪position;
    };
    auto addTriangle = [&](vec3f const &p1, vec3f const &p2, vec3f const &p3) {
        jellyGeometry.push_back(givr::Triangle{p1, p2, p3});
    };

    for (std::size_t i = 0; i < resolution; ++i) {
        for (std::size_t j = 0; j < resolution; ++j) {
            for (std::size_t k = 0; k < resolution; ++k) {
                if (i == 0  && j!=0 && k!=0) {
                    addTriangle(pos(i, j-1, k-1), pos(i, j, k), pos(i, j, k-1));
                    addTriangle(pos(i, j-1, k-1), pos(i, j-1, k), pos(i, j, k));
                }
                if (i +1 == resolution  && j +1 != resolution && k != 0) {
                    addTriangle(pos(i, j+1, k-1), pos(i, j, k), pos(i, j, k-1));
                    addTriangle(pos(i, j+1, k-1), pos(i, j+1, k), pos(i, j, k));
                }
                if (j == 0  && i!=0 && k!=0) {
                    addTriangle(pos(i-1, j, k-1), pos(i, j, k), pos(i, j, k-1));
                    addTriangle(pos(i-1, j, k-1), pos(i-1, j, k), pos(i, j, k));
                }
                if (j +1 == resolution  && i +1 != resolution && k != 0) {
                    addTriangle(pos(i+1, j, k-1), pos(i, j, k), pos(i, j, k-1));
                    addTriangle(pos(i+1, j, k-1), pos(i+1, j, k), pos(i, j, k));
                }
                if (k == 0  && i!=0 && j!=0) {
                    addTriangle(pos(i-1, j-1, k), pos(i, j, k), pos(i, j-1, k));
                    addTriangle(pos(i-1, j-1, k), pos(i-1, j, k), pos(i, j, k));
                }
                if (k +1 == resolution  && i +1 != resolution && j != 0) {
                    addTriangle(pos(i+1, j-1, k), pos(i, j, k), pos(i, j-1, k));
                    addTriangle(pos(i+1, j-1, k), pos(i+1, j, k), pos(i, j, k));
                }
            }
        }
    }
};
```

6.11 Custom Geometry

This type of geometry is here so that you can specify your own geometry. It is quite flexible, with the caveat that you are required to understand how geometry is typically provided to the GPU and manage all of the indices, vertices, normals colours or uv coordinates yourself. It does very little compile time or run time checking. As a result, you are responsible for all aspects of this particular geometry.

NOTE: The renderers that we use assume a few things about the setup of this data.

- vertices are 3 floats.
- normals are 3 floats.
- uvs are 2 floats
- colours are 3 floats.
- indices are 32 bit unsigned integers.

in order to enforce this convention, the parameters for custom geometry are specified as *vec3fs* or *vec2fs* or single *std::uint32_t* for indices.

Also note, that the vertices, normals, uvs, and colours vector must either contain 0 elements or the same number of elements or you risk a segfault from within the graphics driver.

Also note, that if you provide indices, it will be rendered as indexed geometry. If you do not provide indices it will not be rendered as indexed geometry.

NOTE: None of the current styles use the uv coordinates.

6.11.1 Parameters

The *CustomGeometry* is a templated class, which takes givr::PrimitiveType as the template parameter. This template parameter may be set to any of the *givr::PrimitiveType* values:

```
enum class PrimitiveType {
    POINTS,
    LINES,
    LINE_LOOP,
    LINE_STRIP,
    TRIANGLES,
    TRIANGLE_STRIP,
    TRIANGLE_FAN,
    LINES_ADJACENCY,
    LINE_STRIP_ADJACENCY,
    TRIANGLES_ADJACENCY,
    TRIANGLE_STRIP_ADJACENCY
};
```

The *CustomGeometry* class provides lists of *vec3f* for vertices, normals and colours, a list of *vec2f*

```
template <PrimitiveType PrimitiveT>
struct CustomGeometry {
    std::vector<vec3f> vertices;
    std::vector<vec3f> normals;
    std::vector<std::uint32_t> indices;
    std::vector<vec3f> colours;
    std::vector<vec2f> uvs;
}
```

6.11.2 Data

It provides the data you provide to the style.

6.11.3 Example

No examples for this one. The primary reason is that I haven't written a good example for this, but I'll also claim that if you're considering using this type of geometry, then you should be willing to read an existing tutorial on how to setup these sorts of buffers for rendering. The exact format depends on whether it's indexed, which primitive type you are using etc.

6.12 Actual Types

As mentioned in the introduction, we use the C++ `auto` keyword liberally in the example code above. This hides the actual types that are used throughout. This section explains the types a bit more concretely.

6.12.1 Named Parameters

The various parameters that are passed into the geometry are a sub class of the `givr::utility::Type` class which is templated. These classes wrap some other type, like a `glm::vec3` or a `float`. Each of the sub classes are named after the parameter they represent. Each of the instantiation functions for the Geometry operate on these named types. It is the usage of these named parameters which allows us to perform various compile time checking to ensure your code is more likely to run correctly. It also allows us to take the parameters for a given geometry out of order. In other words, the following two examples are equivalent:

```
auto line = Line(
    Point1(-15.0, -15.0, 0.0),
    Point2(15.0, 15.0, 0.0)
);
```

and:

```
auto line = Line(
    Point2(15.0, 15.0, 0.0),
    Point1(-15.0, -15.0, 0.0)
);
```

6.12.2 Instantiation of Geometry

Each of the geometry types has an instantiation function. These functions are what we use in the above example code. Each function takes in a set of named parameters and then ensures the following:

1. All required parameters are specified.
2. No duplicate parameters are specified.
3. Only parameters that are used are specified.
4. The types of the parameters are valid.

6.12.3 Types of the Geometry

The usage of the the instantiation functions means that the type they return does not have the same name as the function itself. These are the type of each of the geometries used in the examples:

```
SphereGeometry sphere = Sphere();
TriangleGeometry triangle = Triangle(...);
QuadGeometry quad = Quad(...);
CylinderGeometry cylinder = Cylinder(...);

LineGeometry = Line(...);
MultiLineGeometry multiLine = MultiLine(...);
PolyLineGeometry<PrimitiveType::LINE_LOOP> polyline
  = PolyLine<PrimitiveType::LINE_LOOP>(...);

MeshGeometry palmTree = Mesh(...);
TriangleSoupGeometry jellyGeometry;
CustomGeometry<PrimitiveType::TRIANGLES> custom;
```

CHAPTER

SEVEN

STYLES

givr provides the following types of styles.

7.1 GL Lines

The line style is used to render the geometry types that are rendered using OpenGL lines. Lines have no normals or shading. They cannot be rendered with phong shading. Thus the `givr::styles::GL_Line` style is necessary.

7.1.1 Parameters

Lines have two parameters:

- `Colour(float, float, float)`: The colour of the line.
  - **Required**
- `Width(float)`: The width of the line.
  - *Default*: `Width(1.f)`

NOTE: Not all opengl implementations allow setting line width to all values. When you set it to a value it doesn't support, the resulting line width is usually the closest supported value. Some implementations only support line width of 1.0. If you need better control over your lines, then consider the cylinder class

7.1.2 Required Data

Lines require that the geometry you provide it uses one of the following primitive types:

- `givr::PrimitiveType::LINES`
- `givr::PrimitiveType::LINE_LOOP`
- `givr::PrimitiveType::LINE_STRIP`
- `givr::PrimitiveType::LINES_ADJACENCY`
- `givr::PrimitiveType::LINE_STRIP_ADJACENCY`

It also requires that the geometry provides vertices.

7.1.3 Example

A simple example:

```
//  make a style that renders lines as green and 15 wide.
auto lineStyle = GL_Line(Width(15.), Colour(0.0, 1.0, 0.0));
```

7.2 NoShading

The NoShading style is a simple style which simply fills in geometry with a single colour. No shading is done. Useful for things like using cylinders to approximate lines in an orthographic view.

7.2.1 Parameters

The NoShading style provides a single parameter

- `Colour(float, float, float)`: The colour of the object
  - **Required**

7.2.2 Required Data

It has no restrictions on the type of geometry it can render.

7.2.3 Example

A simple example:

```
auto noshading = NoShading(Colour(1.0, 0.0, 0.0));
```

7.3 Phong

The phong style is a simple style which provides 3D shaded geometry.

NOTE: If you provide normals that are not smooth, then the phong shader will generate flat shading. As an example, if you use the *TriangleSoup* geometry with the phong shader, it will use flat shading as the normals provided by this geometry are not smooth across adjacent triangles edges.

7.3.1 Parameters

The phong shader provides a number of parameters

- `Colour(float, float, float)`: The colour of the object
  - **Required**
- `LightPosition(float, float, float)`: The position of the light.
  - **Required**
- `AmbientFactor(float)`: The Ambient lighting factor.

  - *Default*: `AmbientFactor(0.05f)`
- `SpecularFactor(float)`: The Specular lighting factor.
  - *Default*: `SpecularFactor(0.3f)`
- `PhongExponent(float)`: The Phong Exponent.
  - *Default*: `PhongExponent(0.8f)`
- `ShowWireFrame(bool)`: Whether to show wireframe or not. Uses the geometry shader.
  - *Default*: `ShowWireFrame(false)`
- `WireFrameColour(float, float, float)`: The colour of the wireframe.
  - *Default*: `WireFrameColour(0.f, 0.f, 0.f)`
- `WireFrameWidth(float)`: The approximate width of the wireframe lines.
  - *Default* `WireFrameWidth(1.5f)`
- `GenerateNormals(bool)`: Whether to automatically generate normals for each triangle. Uses the geometry shader. Normals are per-triangle and as such produce flat shading.
  - *Default*: `GenerateNormals(false)`
- `PerVertexColour(bool)`: Whether to use the colours specified as part of the geometry, where each vertex has its own colour value.
  - *Default*: `PerVertexColour(false)`

7.3.2 Required Data

The phong style requires that your geometry uses one of the following primitive types:

- `givr::PrimitiveType::TRIANGLES`
- `givr::PrimitiveType::TRIANGLE_STRIP`
- `givr::PrimitiveType::TRIANGLE_FAN`
- `givr::PrimitiveType::TRIANGLES_ADJACENCY`
- `givr::PrimitiveType::TRIANGLE_STRIP_ADJACENCY`

It also requires that the geometry provides vertices.

7.3.3 Example

A simple example:

```
auto phongStyle = Phong(
    LightPosition(0.0, 0.0, 100.0),
    Colour(1.0, 1.0, 0.1529)
);
```

CHAPTER

EIGHT

INDICES AND TABLES

- genindex
- modindex
- search