Overview

`libopendrop` “presets” are self-contained computer graphics programs that convert timestamped audio samples into image frames. They are currently implemented very explicitly, in that there is an `opendrop::Preset` abstract class for them. Preset implementations extend this class to provide the conversion explicitly. The presets are required to perform any necessary instantiation and management. Changes made to the preset to explore the parameter space quickly clutter the implementation.

Suspend for a moment the design decisions around `opendrop::Preset`. Within any preset implementation, the statements convert data from one format to another. The implementation is generally modular in an abstract sense, and each “module” is an expression which drives a particular aspect of the visual presentation of the preset.

As an example, suppose we have a preset that shows the super cool effect, `screenbow`! In this effect, the color of the entire screen is a simple function of time. We could implement one possible conversion of this type[1] with this expression:

```
glm::vec4 color = HsvToRgb(glm::vec3(state->time(), 1.0f, 1.0f), 1.0f);
```

In this case, we’re converting a monotonic input (`time`) into a color output (`glm::vec4` containing RGBA).

[1] I mean the literal type of the conversion, when considered as a function: `Output (*)(Input)`

Proposal

We can declare this conversion more explicitly using the following:

```
Opendrop::DeclareConversion<
  libopendrop::GlobalState,
  libopendrop::Monotonic>(
  /*name=*/"time_is_monotonic",
  /*conversion_function=*/[](libopendrop::GlobalState s) {
    return s.time();
  });
Opendrop::DeclareConversion<libopendrop::Monotonic, libopendrop::Color>(
  /*name=*/"color_wheel",
  /*conversion_function=*/[](libopendrop::Monotonic m) {
    return HsvToRgb(glm::vec3(state->time(), 1.0f, 1.0f), 1.0f);
  });
```

With the above, we are telling `libopendrop` two facts:

- A monotonically increasing value can be derived from the current time.
- A color can be derived from a monotonically increasing value by looking around the perimeter of a color wheel according to the value.

`libopendrop` can then use this information to derive logical consequences of the presented set of conversion operators. As an example, suppose we add the following declaration:

```
Opendrop::DeclareConversion<libopendrop::Color, libopendrop::Texture>(
    /*name=*/"solid_single_color",
    /*conversion_function=*/[](libopendrop::Color c) {
      Texture t;
      auto activation = t.ActivateRenderContext();
      GlClear(c);
      return t;
    });
```

With all of the preceding facts, `libopendrop` performs induction to derive this conversion:

```
Opendrop::DeclareConversion<void, libopendrop::Texture>(
  /*name=*/"...",
  /*conversion_function=*/[]() {
    Texture t;
    auto activation = t.ActivateRenderContext();
    GlClear(HsvToRgb(glm::vec3(state->time(), 1.0f, 1.0f), 1.0f));
    return t;
  });
```

Conveniently, when no conversions from `opendrop::Texture -> void` are defined, the only available conversion found is (the default declared):

```
using Attributes = opendrop::Conversion::Attributes;
Opendrop::DeclareConversion<libopendrop::Texture, void>(
  /*name=*/"blit_to_screen",
  /*conversion_function=*/[](libopendrop::Texture t) {
    MagicFunctionThatPutsTheTextureOnTheScreen(t);  // 😁
  }).SetAttribute(Attributes::kOnlyOne, true)
    .SetAttribute(Attributes::kAlways, true);
```

Note that we mark the conversion operator with `Attributes::kOnlyOne` and `Attributes::kAlways`; this is to tell `libopendrop` that there should always be exactly one of these in the graph.

Need to ~~Get Rid of Something~~ Spread Some Awesomesauce?

Suppose that some aspect of the computation of a conversion operator is useful as another output, but should not affect how the conversion operator is considered for induction by `libopendrop`… for this case, we introduce another feature:

```
Opendrop::DeclareConversion<libopendrop::Monotonic, libopendrop::Texture>(
  /*name=*/"shades_of_grey",
  /*conversion_function=*/[](libopendrop::Monotonic m) {
    Texture t;
    auto activation = t->ActivateRenderContext();
    float sinusoid = Opendrop::Consume<opendrop::Unitary>(
      /*name=*/"sinusoid", /*value=*/(1.0f + sin(m)) / 2.0f);
    GlClear(glm::vec4(sinusoid, sinusoid, sinusoid, 1.0f));
    return t;
  });
```

The above conversion makes a solid-color texture that varies from black to white by a sinusoid of the input value. The sinusoid of the input is a potentially useful unitary value that could be consumed elsewhere in the graph. We expose it for use by other graph nodes using `Opendrop::Consume()`. We assign it a name that can be used to refer to it. This name is composed with the name of the conversion it which it appears, resulting in the global-namespace name `"shades_of_grey::sinusoid"`.

Recursion

We can implement the classic background zooming effect with a recursive conversion:

```
Opendrop::DeclareConversion<
  std::tuple<libopendrop::Texture, libopendrop::Texture>,
  libopendrop::Texture>(
  /*name=*/"zoom_recurse",
  /*conversion_function=*/[](libopendrop::Texture prev,
                             libopendrop::Texture input) {
    Texture output;
    auto activation = output->ActivateRenderContext();
    // Zoom the screen inwards by 5% every frame.
    Blit(WarpZoom(prev, /*zoom_coeff=*/1.05f), &output);
    Blit(input);
    return output;
  }).Attribute(Attributes::kRecursive, true);
```

We mark the conversion with the attribute `Attributes::kRecursive`. This tells `libopendrop` that the first argument to the conversion function should be provided with the output value of the invocation performed on the previous graph evaluation.

Here's an example of a recursive kaleidoscopic fractal effect:

```
Opendrop::DeclareConversion<
  std::tuple<libopendrop::Texture, libopendrop::Monotonic,
             libopendrop::Texture>,
  libopendrop::Texture>(
  /*name=*/"kaleidoscope_recurse",
  /*conversion_function=*/[](libopendrop::Texture prev,
                             libopendrop::Monotonic m,
                             libopendrop::Texture input) {
    Texture output;
    auto activation = output->ActivateRenderContext();
    Blit(opendrop::Export(WarpKaleidoscope(prev,
                                           WarpKaleidoscopeOptions {
                                             .angular_offset=m,
                                             .num_petals=6,
                                             .tile=true
                                           })), &output);
    Blit(input);
    return output;
  }).Attribute(Attributes::kRecursive, true);
```

Need a Value You don’t Have?

For this case, we introduce the reverse of opendrop::Consume()... opendrop::Produce(). Use it to materialize any needed values while similarly ignoring these inputs during induction:

```
Opendrop::DeclareConversion<
  std::tuple<libopendrop::Texture, libopendrop::Texture>,
  libopendrop::Texture>(
    /*name=*/"kaleidoscope_recurse",
    /*conversion_function=*/[](
      libopendrop::Texture prev, libopendrop::Texture input) {
    Texture output;
    auto activation = output->ActivateRenderContext();
    Blit(opendrop::Export(WarpKaleidoscope(prev, WarpKaleidoscopeOptions {
     .angular_offset=opendrop::Produce<opendrop::SmoothUnbounded>(),
     .num_petals=opendrop::Produce<opendrop::Count>(),
     .tile=opendrop::Produce<opendrop::Bool>()
    }))), &output);
    Blit(input);
    return output;
  }).Attribute(Attributes::kRecursive, true);
```

Guiding Configuration Space Exploration

As we continue to lay out rules, the configuration space explodes combinatorially. We could allow `libopendrop` to explore this configuration space freeform (and indeed we have this option). However, there's value in having a human architect specify parts of the configuration space that are interesting; we can thus more rigorously define "preset" as a reference to any such configuration.

We can define presets by aggregating conversions by their names into larger structures using a *preset configuration file*. Such a file is formatted as a text protocol buffer with the following contents:

1. a name
2. who the author is
3. an expression that describes the preset behavior (making use of declared conversions)
4. Notes

When libopendrop loads the preset configuration files, it parses them into in-memory opendrop::Preset instances. These instances contain a representation of the specified expression.

The instances can be "instantiated" within the `GraphBlender` by invoking:

```
opendrop::Preset my_preset =
Preset::FromConfigurationFile("my_preset.preset")
graph_blender.Override(my_preset.graph());
```

This does the simple-stupid thing of replacing whatever graph is currently instantiated in the `GraphBlender` with the provided preset graph. The provided graph should generally be of the type `Texture (*)(void)`, but this is not strictly necessary–`libopendrop` will minimally expand the graph to satisfy all input arguments and output `Texture`.

Interpolation

Now the fun part–any two points in the configuration space (any two graph configurations of the conversion operators) can be connected together by a string of other points whose distance from one another is 1.

We define "distance" as Levenshtein distance, although the distance metric has some fractal character to it in that elisions and elaborations where the types are sound are permissible substitutions. What we mean by this is that given these operators:

monotonic_to_color(monotonic) -> color
color_to_tex(color) -> tex
monotonic_to_tex(monotonic) -> tex

The former 2 can be combined to achieve the same signature as the latter, and are thus a permissible *elaboration* of a part of the graph. Similarly, given the following:

<elision example>

Memory Model

As the graph elaborates, we have potential infinite growth in the number of intermediate values that we have to retain in order to be able to evaluate the next frame. We can artificially curtail this growth with an additional weighting mechanism.

In this mechanism, we compute a **cost** associated with depending on a particular value. This cost is proportional to the execution time required to compute the value, and inversely proportional to the number of outputs that currently depend on it.

We additionally modulate the cost by a binary function which evaluates whether or not the value is currently part of the computation at all (no forward dependency paths that end at the output Texture).

Since finding this value is also NP-hard (counting paths between two points in a graph) we can only hope to estimate it heuristically (except in very trivial cases). We can do this by fitting an exponential to the statistics of the problem for graphs of N+E < 100, and use that result instead.

Whenever an output value has no outgoing edges, we can trim the graph starting at that node by evaluating if any of the contributing conversions are exclusively feeding it, in which case they too can be trimmed. We can keep track of multiple such output values and trim the contributing conversions in a trimming pass that considers all outputs in tandem.

Once the value is trimmed, its memory will be automatically freed. In this way, we can make heap space available for continued computation.

Blending

We can define *blend functions*, which are functions of the form:

```
template<typename T>
T Blend(T& a, T& b, float alpha);
```

Note that this looks very much like `Lerp()`, and indeed, `Lerp()` is an example of a blend. However, there are many other ways to smoothly transition between two 1D float values (spline interpolation, sinusoidal curve easing, etc.). Likewise, there are many possible blending implementations for other types as well. As an example, two textures could be blended linearly by treating them as 3D vectors and doing the blending element-wise. However, it would make for a much more impressive effect if the blending is done using a feature-matching cage deform + lerp method.[2]

Alternative Implementations

Exporting/Importing Values

Another way to implement this is to simply have all inputs and outputs of the function be substituted independently of one another, and have user-specified weights for each. I think that's equivalent…

Another attribute that should probably be included is a name for the value (whether input or output). The names could also be used to logically group values, but I think that's more appropriately done with a new type.[3]

We can then automate the import and export of particular values using regex matching on the port names to connect new graph segments. As an example:

---

[2] http://www.morpheussoftware.net/

[3] Adding types could be simplified with some sort of template singleton… which could be achieved using a Python preprocessor extension. Or we could add a macro.

Button -> Graft(FindValue(“.*radius.*”), FindProduction(“midi_input”))

`Graft` is of course short for `Blend(/*duration_s=*/0.0f, …)`.

Recursion

Recursion could also be achieved with a more rich interface by adding a distinct builder method:

`Recurse(absl::Span<int> input_mapping)`

With this method, we can enable the user to specify which output values from the function should be stored and recalled as inputs in the next graph evaluation. [4]

History

Now the interesting stuff… suppose

Musings on Effects and How to Achieve Them

Speckling

On a button press, add a number of texture override grafts and start cycling their blends back and forth at random phases. Modulate the speed of cycling with a knob.

As an aside, why not associate knobs and buttons into composite inputs in the order in which the buttons are pressed?

[4] We can further snapshot a particular state of the execution graph and restart evaluation from that point. The maximum recursion depth will expand the amount of input that we have to capture in order to reproduce the effect cleanly. We can trigger this replay behavior on an oscillator modulated by a user input (like a button).