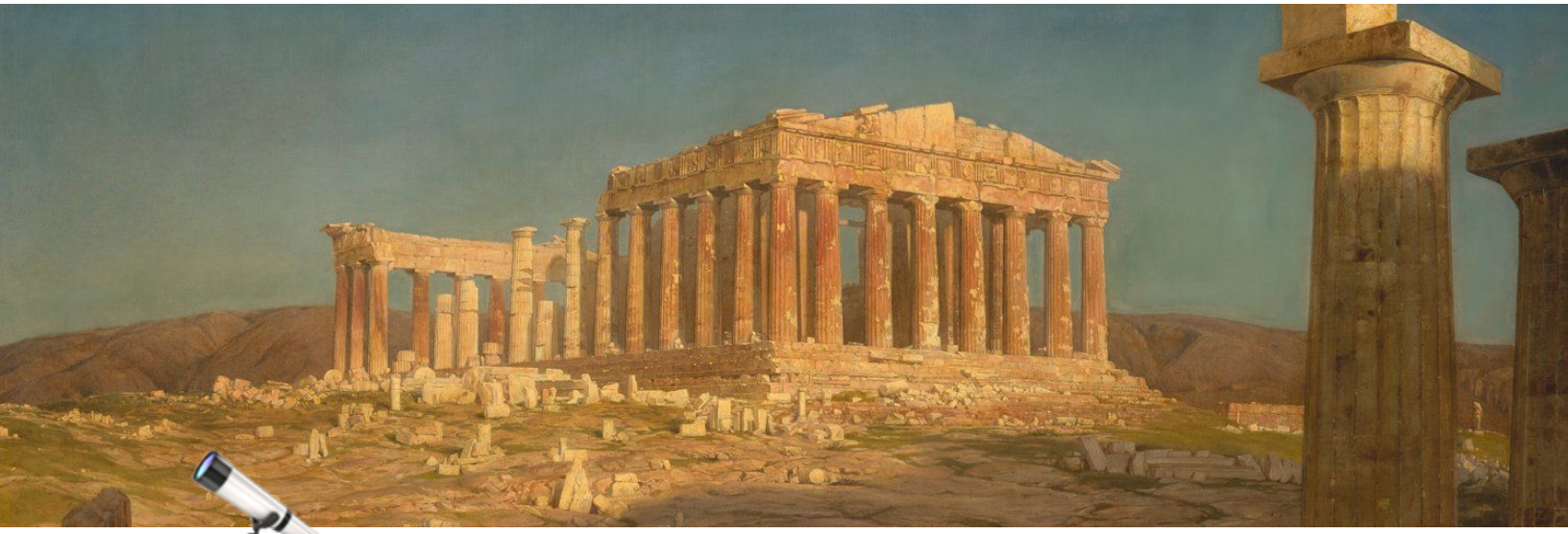


CSE 501 - Final Report

Outline

1. Motivation
2. Brief Tutorial
3. Compiler pipline, AST, IR, Running example
4. "Smarts" of compiler, analysis, synthesis, or optimization
5. Experimental Evaluation
6. Reflection
    1. Lessons learnt: what would you do differently if you could redo the project?
    2. What knowledge and skills you wish you had when you started?
    3. What technologies do you wish you could build on in your project?

Motivation

Nowadays there are great resources for creating art using computers. The `processing` and `p5.js` languages provide an easy abstraction layer over OpenGL that allows for quick prototyping while supporting a wide array of interesting visuals in just a few lines of code.

However, in one of the oldest computer art subcultures, the demo-scene, a different rendering technique is often used to great effect. The ray marcher algorithm is written entirely on the gpu and has many great properties that have allowed coders to fit amazing visuals in just a few kilobytes for decades. For newcomers though there are few resources available to use this technique as it relies on a deeper understanding of the graphics pipeline, mathematics, and the poorly documented glsl language. These are some of the problems this language seeks to help.

- Fully featured standard library with many shapes and operations available.
- Strong performance characteristics.
- Easy debugging and visualizations.
- Human readable compiled code.
- Fully extendible at all points of the language.

Tutorial

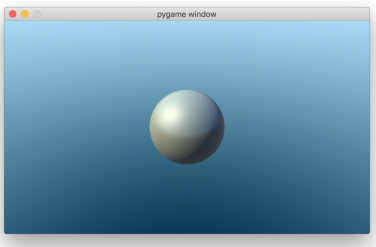
Objects

Objects represent the main structure of a program. They can be used to layout a scene or to define a larger structure to be used elsewhere. To define an object, we create a python function and use the `@Object.register` decorator to tell the compiler we want to treat this function as an object.

First an example of the simplest program in `basic.py`:

```python
# basic.py from marcher.march import *
@Object.register() def Basic(self):
self.res(Union, Sphere(2.0))
Camera((600, 350),
AA=2).view(Object.Basic)
```

Python

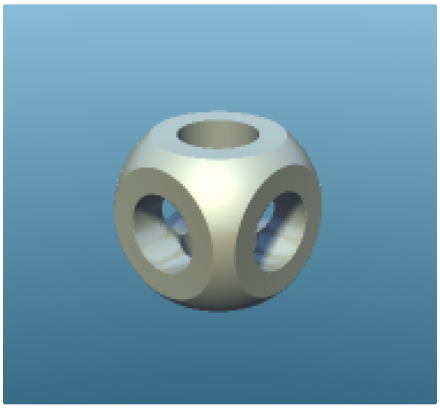


This will produce a single floating sphere as expected.

The following is a short working example file contained in `object.py` :

```python
# object.py from marcher.march import *
@Object.register() def MyObject(self):
self.res(Union, Box(vec3(1, 1, 1)))
self.res(Intersect, Sphere(1.3)) r =
0.5 l = 2 self.res(Subtract,
CylinderX(vec2(r, l)))
self.res(Subtract, CylinderY(vec2(r,
l))) self.res(Subtract,
CylinderZ(vec2(r, l))) c = Camera((650,
380), AA=2) c.save(Object.MyObject,
"gen.glsl") c.view(Object.MyObject)
```

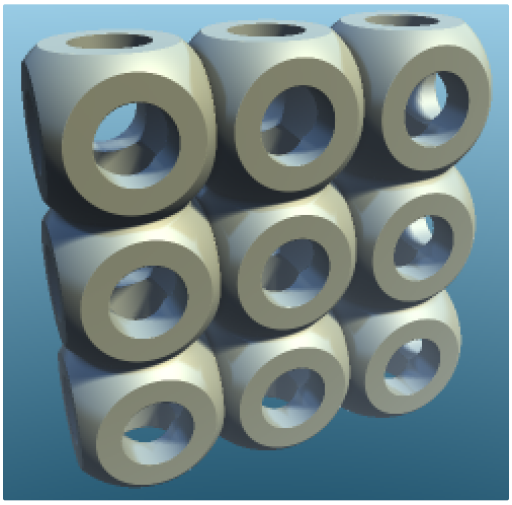
Python



Several things are happening here. First the `Camera` object is what defines some rendering parameters and is what allows us to view our compiled code. We'll touch more on this later. The main way to use Objects are with the `self.res` command which takes a Combinator as the first argument and any Primitive or Combinator as the second. We can also use Other Objects as they count as Primitives:

```python
# object.py @Object.register() def
MyObject(self): ...
@Object.register() def
MyScene(self): mo = MyObject() for
i in range(-1, 2): for j in
range(-1, 2): pos = 2 * vec3(i, j,
0) self.res(Union, mo.at(pos))
c.view(Object.MyScene)
```

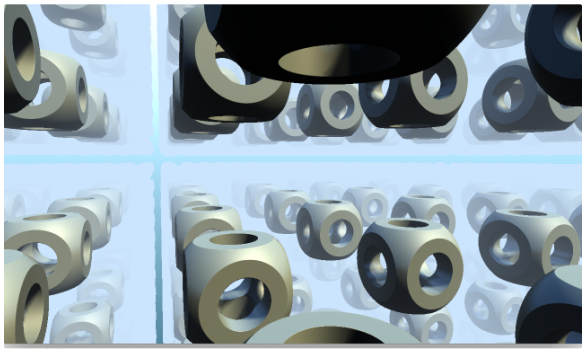
Python



The second function in an Object is `self.p` . This takes any Operator and applies it to the current space meaning all `self.res` calls afterwards will have this changed space. A good example is the `Repeat` Operator which mods space:

```python
#infinite.py
@Object.register() def
Infinite(self): r = 4.5
self.p(Repeat(vec3(r, r, r)))
self.res(Union, MyObject())
c.view(Object.Infinite)
```

Python

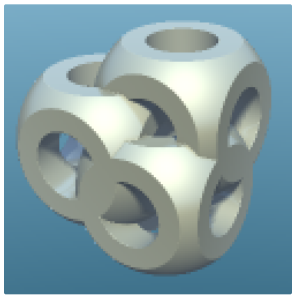


Combinators

The three basic combinators are Union, Intersect, Subtract. We saw them being used inside of an Object but they are more commonly used to build up Objects without many `self.res` calls. Here we rewrite the Object example using only Combinators and regular python functions.

```python
def make_object(): shape = vec2(0.5, 2) cross =
Union(CylinderX(shape), Union(CylinderY(shape),
CylinderZ(shape))) rounded_box =
Intersect(Box(vec3(1, 1, 1)), Sphere(1.3))
return Subtract(rounded_box, cross)
@Object.register() def MyScene(self): obj =
make_object() self.res(Union, obj.at(vec3(1, 1,
1))) self.res(Union, obj.at(vec3(1, 2, 1.5)))
self.res(Union, obj.at(vec3(2, 1.5, 1)))
```

Python



Here we are also using the `at` keyword which we will cover more under Primitives

Operators

Operators are any function that takes in a space and outputs a transformed space. We call these 'operators' because they can have drastic effects on Objects. The most common Operator is `Translate` which shifts everything by the input vector. In fact, this operator is so common that all objects have an `at` keyword that corresponds to a final translate by the given vector.

This can either be set on creation or later called to move an already defined objects

```python
Sphere(1., at=vec3(1, 1, 1)) == Sphere(1.).at(vec3(1, 1, 1))
```

If we want to specify an arbitrary Operator we use the `f` keyword which can also be used at creation or later:

```python
Sphere(1., f=Translate(vec3(1, 1, 1))) == Sphere(1.).at(vec3(1, 1, 1))
Sphere(1., f=Translate(vec3(1, 1, 1))) ==
Sphere(1.).f(Translate(vec3(1, 1, 1)))
```

Because the input and output of Operators are both "spaces" they lend themselves well to chaining. One simple way to achieve this is with an Operator's own `f` call. However, there is also a `*` syntax for composition:

```python
t1 = Translate(..., f=Translate(..., f=Translate(...))) t2 =
Translate(...) * Translate(...) * Translate(...) Sphere(1.).f(t1) ==
Sphere(1., f=t2)
```

Primitves

These functions are the leaves of the recursive calls. They take a space and parameters and output a single distance. A Common example has been the `Sphere` function. Because Objects are Subclasses of Primitives, they can be used almost inter changeably. However, the method for defining them is slightly different although similar to Objects:

```python
@Primitive.register() def Sphere(p: vec3, r: float) -> float: """
return length(p) - r; """
```

- `@Primitive` : This tells the compiler we want to register a new Primitive
- `register()` : Here we specify dependencies, the compiler can only infer them for Object
- `Sphere` : This is the name to register the function under
- `p: vec3, r: float` : These are the parameters the function takes
  -

    - Primitives take `p: vec3`
    - Combinators take `d1 float, d2: float`
    - Operators take `p: vec3`
    - Objects take `` `p: vec3, res: float``
- `-> float` : This is the function output, it is not required for the basic function types
    - Operators are the only function that does not output `float`
- `return length(p) - r;` : This is the body of the function, it is what will eventually be compiled into the shader
    - It is possible to call Objects from primitives as long as they are specified in the dependencies

Advanced features

- [ ] Objects with parameters
- [ ] Function
- [ ] Calling Objects in Primitives
- [ ] Var
- [ ] Vec
- [ ] Camera

Algorithms

The bulk of the program is in the definition of the distance estimator function. This function takes in a point in space and outputs its minimum distance estimation in the form of a float.

The first tricky part is in the Object definitions. These are python functions that define roughly what the corresponding glsl function should look like. However, one major limitation is that we cannot allow variables or loops in the output. We can also only run each function once to compile it and we only want to run the ones we need if we actually need them.

The first trick is to decorate the function with a python class that stores the function. Next when the function needs to be compiled, the class calls the function and passes itself as an accumulator. The result is then memoized inside

the class. The effect of executing the function is both the linearized calls within the function as well as all the dependencies

The next trick is how we compile the entire program. We have a starting function and for each function a set of functions it depends on. To solve this problem we use a toposort to get a list of functions in the order they are required.

- [ ] Partial evaluation
- [ ] Operator composition
- [ ] Dependency graph
- [ ] Dependency graph creation
- [ ] Lazy evaluation of object bodies

Evaluation

From the testing I've done, the result of removing all loops and variables increase the average fps by 5-10 for certain scenes. However, this improvement is almost certain to change based on the graphics card and drivers used to run the shader.

The much better improvement is on ease of programming and abstraction. While testing the compiler I was able to quickly create examples that had a wide range of functionality. The syntax meshes nicely with python and allows for easy expression.

Reflection

While I did change to this idea later, I was able to add most of the ideas I had. One major ability I wanted for the language was being able to write definitions and have them automatically generate the needed python code. This also allowed me to have the output code be fairly human readable. This helped a ton with debugging.

Next steps

There were a few major things I wanted to include that aren't possible yet.

First having a Scale Operator is difficult and cannot be done in the normal way. This is because by multiplying the space, we need to divide all distance that use

the stretched space. This currently cannot be supported by the function templates as it would require a sort of first class function that is not in glsl
The next big feature would be per Primitive or per Object materials. First this would take the form of choosing different colors but could be extended to many different effects such as transparency, reflection, glow, and many others. I have some ideas for how to achieve this but it will require some major rewrites.