See++ Project Proposal

By Adar Tulloch, Vishrut Tiwari, Winston Zhang, and Jack LaVelle

Intro:

Our project will be a language capable of making graphics. Starting with pixels as the base building block users will be able to build lines, shapes, 2D custom graphics, and more. They can combine with other user made models in order to create “computer graphics art” and build on previous projects. Our project will be largely inspired by the programming language Processing and we will follow the object oriented programming paradigm mixed with the simplified Java like syntax.

Technologies:

This section is a result of a preliminary search into what technologies we will need to use. It is difficult to know exactly what we need to incorporate in this project and exactly by what means we will go about it. For example, it is obvious that we will need some sort of version control that will let us function as a group where we can independently work on separate parts of the project simultaneously. These are the things we know we do not know. But it is the things we do not know we do not know about that may or may not make up a large part of this project. Other projects have to deal with this aspect sure but it is the graphical side of things in See++ that will make it more interesting for us.

For now this is what we plan: we will use BitBucket to host our repository and `ocamllex` to help us scan our input and we can also take advantage of the built-in regular expressions. Then we would use OCaml 4.13.0 for general parsing and AST components. Finally OpenCL will also be helpful for C style function calls. To help us write low level code as we will not be generating machine code directly is LLVM in C++ (no, not See++).

Key Preliminary Custom Data Structures and Types of See++

The underlying graphic data structure among all others will be pixels from which the others are built. For example lines and shapes (with or without fills) will be arrays or matrices of pixels. The various functions such as `vertex(x,y)`, `line(`x_1`,`y_1`,`x_2`,`y_2`)`, or `rect(a,b,c,d)` will effectively generate these points/arrays/matrices of pixels although the programmer will not deal with these directly (as that would be annoying … for example if the programmer wants to make a line with a set color and then change that color they would not want to change the color of every pixel but rather deal with the line as a whole).

Something else we plan to incorporate are mask types which can be applied to existing objects that can modify them. For example a shader mask applied to a shape would decrease the color (that is make it darker) of the whole shape. These masks would not necessarily need to act on the shape uniformly and if we were more optimistic we would claim that perhaps we could incorporate a sophisticated lighting mask that could modify each pixel in the 2D space depending on objects designated as light sources. However, we say this without any idea of how complex these systems would be or how difficult it would be to implement them. This may be achieved using simple convolutional matrices but might need more than just a blur or sharpen matrix to yield this effect. Still one can get a basic understanding of the things we aim to accomplish with this language from this kind of talk. Another example of this would be a smoke mask that, when applied to an object, has a probability of changing every pixel of the shape into grey/white to make it look like that object is shrouded in smoke. This probability distribution would be nonuniform, decreasing from the point of the object designated as the center. These masks could act on multiple objects at once with the space that the probability distribution acts on being the union of the set of coordinates belonging to the objects. Multiple masks could also act on objects as well.

What makes this project special, well what we believe makes it special anyway, is the fact that it would not only act as a way for the programmer to build a very foundational structure for their art (with basic shapes and lines) but even go so far as to provide a platform for rudimentary visual effects. Furthermore,

we aim to make building custom shapes with custom effects as easy as possible to reemphasize the fact that our language serves as a simple but effective tool for providing the means to create much more complex art than what the default shape types alone would provide.

Language features:

Primitive Data Type	Storage
Integer	32- bit
Pixel	Unsigned 8-bit Integer
String	List of characters (stored in heap)
char	16 bit
Double	64 bit
Float	32 bit
Boolean	16 bit
Pixel[$(r_1, g_1, b_1), (r_2, g_2, b_2), .. , (r_{wxh}, g_{wxh}, b_{wxh})$]	Contains rgb values of all pixels in the canvas. So if canvas was 100 by 100 then 10,000 pixels would be in the in array

Functions	Description
`vertex(x,y)`	Places a point on the canvas where `x` and `y` are the coordinates
`strokeWeight(weight);`	Takes in a float which is the weight of the line
`stroke(r,g,b);`	Outlines an object's stroke color with provided red, green, blue pixel values
`fill(r,g,b)`	Set the red, green, and blue color values of a

	canvas which have to be an integer between 0-255
line(x_1,y_1, x_2,y_2)	Draws a line with two coordinates
triangle(x_1,y_1, x_2,y_2, x_3,y_3)	Draws a triangle with float as input for position and three coordinates
rect(a,b,c,d)	Draws a rectangle with float as input for position (a,b), one coordinate, width (c), and height (d)
quad(x_1,y_1, x_2,y_2, x_3,y_3,x_4,y_4)	Draws a quadrilateral with float as input and four coordinates
ellipse(a,b,c,d)	Draws a ellipse with float as input for position (a,b), one coordinate, width (c), and height (d)
arc(a,b,c,d,start,stop)	Draws an arc as input for position (a,b), one coordinate, width (c), and height (d), and the angles to start and stop the arc in radians
size(width, height)	Sets the dimensions of the display window/canvas where users can visualize their output
background(r,g,b)	Set the red, green, and blue color values of the canvas which have to be an integer between 0-255

Sample Code:

```
void setup() {
  size(640,360);
  background(51, 0, 0);
}
void draw() {
  fill(102);
  stroke(255, 0, 0);
  strokeWeight(2);
  vertex(0, -50);
  vertex(14, -20);
  line(0,-50, 14,-20)
  vertex(47, -15);
  vertex(23, 7);
  line(47,-15, 23,7)
  vertex(29, 40);
  vertex(0, 25);
  line(29,40, 0,25)
```

```
  vertex(-29, 40);
  vertex(-23, 7);
  line(-29,40, -23,7)
  vertex(-47, -15);
  vertex(-14, -20);
  line(-47,-15, -14,-20)
}
```

The `setup()` method only runs once and it allows the users to use the `size()` function seen before and the `background()` function to set up their output display. Note that any variables declared inside the `setup()` method will not be global and can not be used inside of `draw()`. The simple `draw()` method allows users to make their own custom object, set the background color, and choose where in the canvas they want to place their object. Within `draw()`, users can make calls to functions such as `fill()`, `stroke()`, and `strokeWeight()` to set up the details of the graphic and call `vertex()` repeatedly to implement the star-shaped figure in this example.

Scheduling and Roles:

We plan to take proactive approach in regards to planning the project in that we expect to meet once or twice a week to discuss any problems we have/will have (especially considering a team members work may rely on someone else and they may not be able to start work until the other has finished / knows exactly what they are dealing with).

The original roles planned for us are:

Manager (the boss): Adar

Language Guru - Winston

System Architect - Jack

Tester: Vishrut

We expect there to be much flexibility and overlap among those roles … but to what extent we are not sure. We will work on Wednesday's one to three as a group and whoever does not show up pays the rest of the group 20 dollars.

Timeline:

October 11th: Create project BitBucket

October 25th: Have scanner done

November 1st: Create documentation for what we want our grammar to look like

November 15th: Finish Abstract Symbol Tree, Parser, and Environment (interpreter)

December 1st: Successfully generate low level code with LLVM

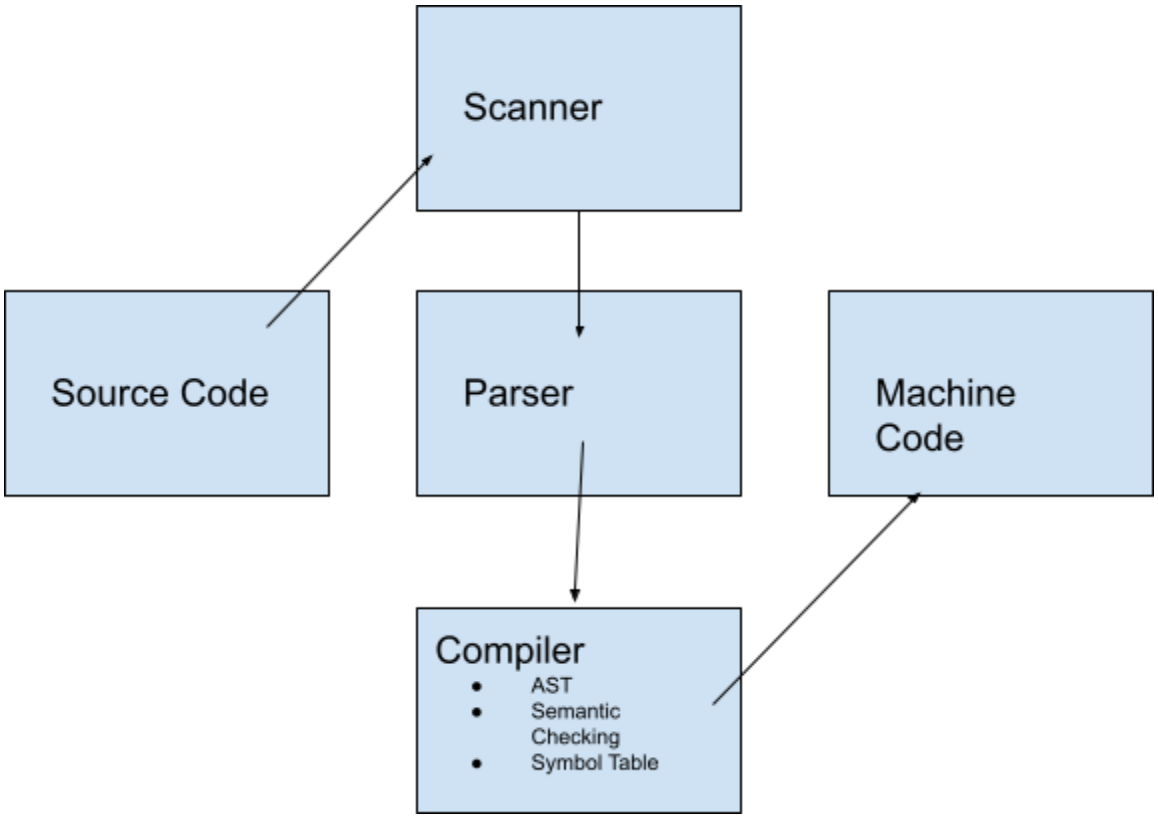
December 5th: First Hello World program

December 15th: Optimize code and create class presentation

Architectural Design:

Source Code will be fed into a compiler, which will then generate the machine code we need. Instead of generating a customized bytecode, we will be able to go straight from the compiler to the LLVM. This mechanism allows us to efficiently run our program. The parser and the scanner will work together before passing the necessary information to the compiler.

Specifically, the source code will be fed into a pre processor before feeding into our scanner. Then the scanner will parse through and give the result to the parser. Finally, the code can be applied and generated by the compiler. In detail, the compiler will contain an AST, which is the first layer and then using both the symbol table and semantic checking, we can finally derive our LLVM code.



Citations:

http://www.cs.columbia.edu/~sedwards/classes/2014/w4115-fall/reports/Sheets.pdf

https://processing.org/