Installing/Using EaselSL

1. You should already have Visual Studio 2010, 2012, or 2013 installed.
2. Copy “SL Game Engine.exe” and “gameTemplate.sl” to your computer. Ideally to somewhere like “C:\Easel\” because sometimes windows will not run an .exe that is elsewhere.
3. Run “SequenceL_Win_64.exe”. This will install SequenceL 2.0 on your computer with a 30-day trial.
4. You should now be able to run “SL Game Engine.exe”, select the file “gameTemplate.sl”, and the game should compile, and then run.

When using the engine, you can run a game by either:

1. Running “SL Game Engine.exe” and choosing an easel game file in the File Dialog.
2. Dragging an easel game file and dropping it on “SL Game Engine.exe”.
3. Running “SL Game Engine.exe” and clicking cancel on the file dialog window. This will run the last game file that was compiled.

Framework Overview

Types

```
5   //region Types
6
7   Point ::= (x: int(0), y: int(0));
8
9   Color ::= (red: int(0), green: int(0), blue: int(0));
10
11  Image ::= (kind: char(1), iColor: Color(0), vert1: Point(0), vert2: Point(0), vert3: Point(0), center: Point(0)
12             radius: int(0), height: int(0), width: int(0), message: char(1), source: char(1));
13
14  Click ::= (clicked: bool(0), clPoint: Point(0));
15
16  Input ::= (iClick: Click(0), keys: char(1));
17
18  //endregion
19
```

The types are present in every Easel game, they are instances of SequenceL's struct types.

A *Point* is a struct with two members, *x* and *y*, which are both integers. A *Point* can be thought of as the usual point in 2D cartesian space, with *x* and *y* being the x and y coordinates respectively. For example: `(x: 500, y: 400)` is a *Point* which will be in the center of the 1000x800 screen.

A *Color* is a struct with three integer members: red, green, and blue. The members of a Color correspond to the RGB color schema used so frequently. For example: `(red: 255, green: 0, blue: 255)` is a *Color* which will appear to be violet once on the screen.

An *Image* is a struct with many members, however most of them will not be used at the same time. In Easel, an image can be a disc, circle, filled triangle, segment, text, or graphic. The *kind* member of the *Image* struct determines which type of image it is, and certain types of images only use certain members of the whole struct. A complete listing of which members of *Image* are used by each image type will be given in the next section but for now, consider the following:

A filled triangle is an image, and it consists of vert1 (*Point*), vert2 (*Point*), vert3 (*Point*), and iColor (*Color*). vert1, vert2, and vert3 represent the three points in space that define a triangle, and iColor is the color that will fill in the triangle.

A text is an image consisting of: center (*Point*), height (Integer), iColor (*Color*), message (*String)*.
Center is the point on the screen where the text will be centered, height is the height of the displayed text, iColor (the name is due to a requirement that names must be unique) is the color of the displayed text, and message is the text to be displayed.

Assume p1, p2, and p3 are defined as *Points*, and *red* is defined as (red: 255, green: 0, blue:

0).
(kind: “triangle”, vert1: *p1*, vert2: *p2*, vert3: *p3*, iColor: *red*) is an *Image* of type filled triangle.
(kind: “text”, center: *p1*, height: 15, iColor: *red*, message: “Hello!”) is an *Image* of type text.

Both of these are *Images*, yet they only use the relevant parts of the type *Image*. Once again, recall that though one can define instances of these types explicitly as in these examples, the next section will go over the Constructor-functions that will make it easier.

A *Click* is a struct consisting of *clicked* (boolean), and *clPoint* (*Point*). *Clicked* tells wether or not a click actually happened, and *clPoint* is the point at which the click occured. An example of a *Click* is not provided, because the developer should never be writing a concrete instance of one unless for testing purposes.

An *Input* is a struct consisting of *iClick* (*Click*), and *keys* (string). The way the user interfaces with Easel games is through th *Input* struct, the click portion determines if/where the user clicks, and the string portion keeps track of which keys are being pressed. An example of an *Input* is not provided, because the developer should never be writing a concrete instance of one unless for testing purposes.

Helpers

```
point(a(0), b(0)) := (x: a, y: b);
color(r(0), g(0), b(0)) := (red: r, green: g, blue: b);
click(cl(0), p(0)) := (clicked: cl, clPoint: p);
input(cl(0), k(1)) := (iClick: cl, keys: k);
segment(e1(0), e2(0), c(0)) := (kind: "segment", vert1: e1, vert2: e2, iColor: c);
circle(ce(0), rad(0), c(0)) := (kind: "circle", center: ce, radius: rad, iColor: c);
text(mes(1), cen(0), he(0), c(0)) := (kind: "text", center: cen, height: he, iColor: c, message: mes);
disc(ce(0), rad(0), c(0)) := (kind: "disc", center: ce, radius: rad, iColor: c);
fTri(v1(0), v2(0), v3(0), c(0)) := (kind: "triangle", vert1: v1, vert2: v2, vert3: v3, iColor: c);
```

The constructor-functions allow the developer to create instances of the types listed above. Note that this method can be used for any user-defined types (for example a *Ball* could be a position (integer) and a velocity (float)).

```
point(100, 300)
```

Will create a *Point* with x: 100, y: 300. This is equivalent to using

```
(x: 100, y: 300)
```

Similarly, color(255, 0, 0) will create a color which looks like pure red, click(true, p1) will create a click. The constructor-functions point, color, click, and input all simply give syntactic shorthand for the sequenceL struct delcaration.

Images:
The constructor-functions that create *Images* are different, however, in that they take only the relevant information as arguments.

Let's say we want to make a segment from *Point* p1 (0,0) to *Point* p2 (200, 200), and we want it to be green. To do this manually we would need to do:

```
(kind: "segment", vert1: p1, vert2: p2, iColor: green).
```

This can be replaced with the constructor-function call

```
segment(p1, p2, green)
```

The remainder of the constructor-functions for *Images* are the same, they are functions that take the relevant members of the *Image* struct as arguments.

Colors:
There are also 9 constants which are very commonly used colors: Black, Red, Orange, Yellow, Green, Blue, Indigo, Violet, and White.

```
dBlack := color(0, 0, 0);
dRed := color(255, 0, 0);
dOrange := color(255, 128, 0);
```

Random:
There is a helper function for generating random numbers, and it works as follows. If the developer requires *n* random numbers, and there is some seed *s* then randSeq(*s, n*) will return a sequence of *n* random numbers [0, 1).

Easel Functions

```
//=================Easel=Functions================================================

State ::= (time: int(0)); //Fill in this struct with the game state members.

initialState := (time: 0);

newState(I(0), S(0)) := (time: S.time + 1);

sounds(I(0), S(0)) := ["ding"] when I.iClick.clicked else [];

images(S(0)) := [text("Time: " ++ Conversion::intToString(S.time / 30), point(500, 400), 30, dBlue)];
```

The fundamental part of an Easel game is in the description of the state of the game, how the state changes, and how parts of the state are displayed to the screen.

State

The *State* struct is going to be different for each game, if a game has particles on the screen then the state must accommodate them. So when the developer creates a game, even from the template, they must define what the state needs to be. For example, if we were trying to recreate pong with a simple scoring system the state might look like this:

```
State ::= (ballPos: Point(0), leftPos: Point(0), rightPos: Point(0),
leftScore: int(0), rightScore: int(0))
```

Note, each game's state must be "State", and user defined structs (such as *Point*) are case sensitive.

ballPos will track the ball's position, leftPos and rightPos will keep track of the paddles for each player, and leftScore/rightScore will keep track of their scores.

Initial State

The initialState function will describe what the beginning state of the game will be, continuing the example from before we may have initialState as:

```
initialState := (ballPosition: p0, leftPos: p1, rightPos: p2,
leftScore: 0, rightScore: 0)
```

Once again "initialState" is case sensitive and required.

New State

This is where most of the work of the developer will take place, this is where the ways in which the state changes are described. Depending on the input, the result of *Input* I in *State* S will be the result of newState(I, S).
If we continue our example with our pong example and encode the rule that says left player's bar moves up when they press "W" it may look like this:

```
newState(I, S) := moveLeftUp(S) when pressed(I) = "W" else S;
```

This says that when "W" is pressed, the resulting state will be what happens when left's paddle is moved up. Otherwise, the resulting state will be the same state i.e. nothing will change.