ART: Animation Rendering Tool

Brett Jervey | baj2125
Gedion Metaferia | gym2103
Natan Kibret | nfk2105
Soul Joshi | srj2120

COMS W4115 Programming Languages and Translators

Programming Language Proposal

September 28, 2016

Overview of ART

Animation Rendering Tool (ART) is an imperative static programming language that allows the creation of 2D animations. ART will use either OpenGL-OCaml bindings or OCaml's built-in graphics library in order to render and update the visual representations of objects. ART will have, at least, limited object orientation support. We plan to design the syntax of ART to be simple and light, similar to 'C' syntax. The proposal below outlines how we envision ART will function; these plans are open to change.

What sorts of Programs are meant to be written in ART?

We envision 2D animation programs to be written using ART. Developers utilizing ART will control where objects are drawn and how their movements are simulated(updated). Below are some examples of how ART can be used.

1. **To model the planetary system:** with the help of ART, a developer can simulate the solar system as well as any hypothetical situation where the physical properties of the sun and the planets (such as mass) are different.

2. **As an interface to simulate properties of physics:** If we consider a physics professor that is trying to simulate the motion of a free falling ball with given mass and gravitational pull, said professor can use ART to simulate the motion of the ball so their students can see the laws of physics in action.

The Parts [and what they do]

1. **Abstraction:**
    a. *Naming via variables*: Developer can name values and objects using strings, such as 'x=1', 'panda = Circle [size]', etc.
    b. *User-defined shape types*: Developer can create new types of shapes by defining the number of edges and vectors as well as the arrangement. They can then create objects using their defined shape. However, the will be an existing library of basic shapes.
2. **Common Patterns:**
    a. *Shapes*: Shapes will be defined as object types.There will be a library of predefined shapes .User can create new shapes.
3. **Control Structures:**
    a. *Loops*:
        i. *General Purpose Loops:* Support for at least one general purpose looping structure, such as a 'for loop', or 'while loop'.
        ii. *Specialized Animation Loops:* ART will provide animation-specific loops, such as a 'Time Loop' (or 'Update Loop')', which simulates the passing of units of time in the animation, and updates all the objects in the program at the end of each iteration of the loop. Such specialized loops are designed to allow users to create animations in a simple and efficient manner.
    b. *Conditionals*:
        i. *If Else:* ART supports 'If .. Else' conditionals similar to 'C'.

4. **Functions:**

   ART supports ‘C’ style user-defined functions. These functions will have user-defined return types. ART also supports functions with no return value (‘void’ return type). Users specify the the arguments, including their type, on function declaration.

5. **Operators:**

   ART will support the standard arithmetic operators(+, - *, /) for its math types with no overloading. For the unique vector type we plan to support standard vector functions like dot product, magnitude using operator signs we have not decided on yet. Our shape functions are only going to have an operator to allow for linear transformation.

   The math and vector types will have standard comparison operators like >, <, >= and ==. Shapes will only have an equality operator for comparison because the user will be able to make more meaningful comparisons by comparing the shape’s internal variables.

6. **Primitives:**

   The primitives of our language can be separated into math types and shape objects. Our math types would include integers, floats and a unique integer or double tuple which we call a vector. These form the backbone of the more arithmetic operations of our language.

   The shape primitives would be shapes such as lines, points, circles, triangles etc which would form the base for user created shape objects or stand on their own. All shape objects will have their own variables which, at creation, would be their coordinates and variables pertaining to their drawing (color, size, etc) with more specific fields for some shapes(ex. radius for a circle). The user will have the ability to add their own custom variables into the shape object like mass or velocity which would be one of the inbuilt math types.

   We would also have a builtin list data structure with a possibility of others data structures as the languages evolves.

7. **Syntax:**

   The ART language syntax draws from the imperative tradition of C and related languages. An imperative coding style is necessary for writing animations in which the programmer has fine tuned control over the components within. ART also contains some object oriented features to allow the definition of user defined shapes. The constructs for function definition, variable declaration, etc. closely mirror that of C. However, in situations where more convenient and intuitive syntax can be applied, we have dropped the C equivalent. For instance, vector norms, tuple assignments etc follow a more pythonic coding style (See Example). There also special naming conventions for core functions that come with the language.

8. **User Inputs:**

   The core language doesn't provide facilities for accepting user inputs as programs written in ART are intended to run mathematically modelled simulations. If time permits however, we hope to add extensions for file IO and keyboard access. The file extension would allow programs to read definitions of shapes specified in configuration files. And Keyboard access would allow users to control and tweak the properties of shapes being animated. This is also enables users to write basic 2D games.

Sample Source Code

Below is a source code that we think will resemble ART's syntax and functionalities. We have also included an example of how the rendered animation will look like.

```
================================================================================
=
void mystepper(Shape obj, Vector acc, double dt):
      obj.x += obj.v*dt
      obj.v += acc*dt

double grav_acc(Vector loc1, Vector loc2):
```

```
      return G/|loc2-loc1|**2

main:
      Disc earth = {r = 1, color=blue}
      Disc sun = {r = 3, color=yellow, fixed="true"}
      $background_color = “black”

      sun.loc = (0,0)
      earth.loc = (10,0)
      earth.v = (0,2)

      timeloop t = 0; t < 40s; dt = 0.01:
            acc =  grav_acc(sun.loc, earth.loc)
            $updateAll(mystepper,acc,dt) # Apply mystepper to all objects
```

==========

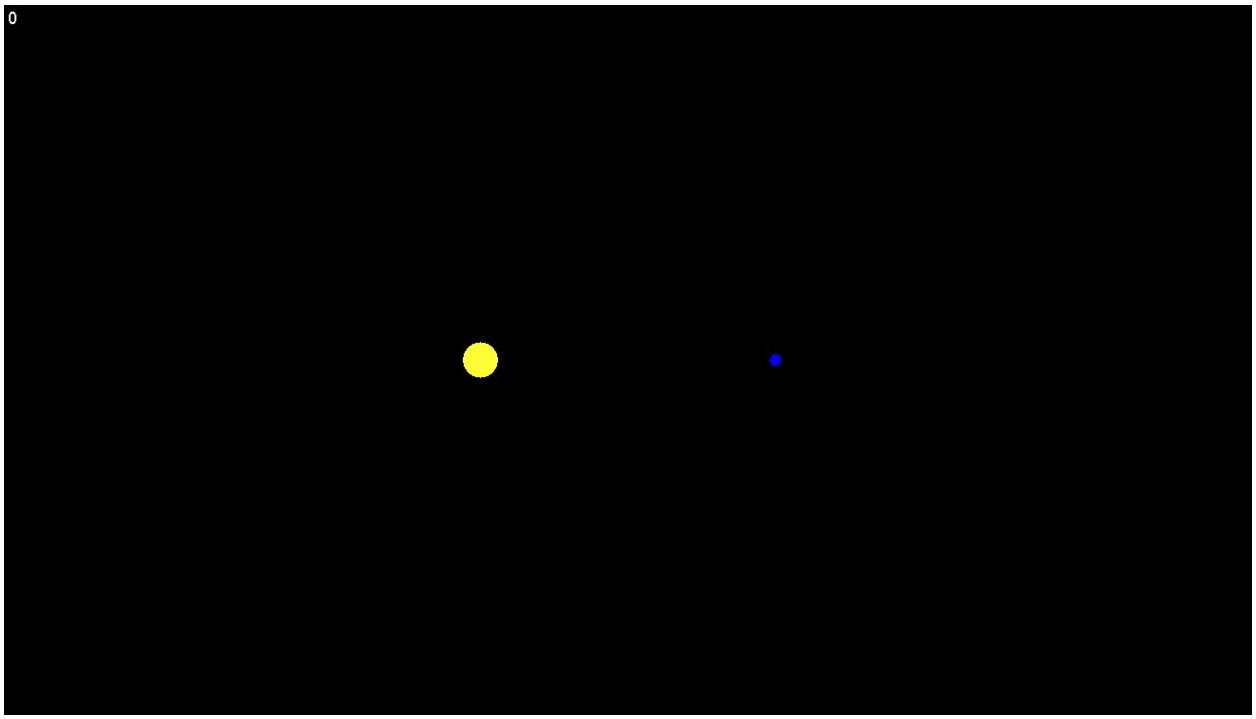

Figure 1 - shows the visual rendition of Earth (Disc earth) and Sun (Disc sun) before the motion begins.