**plt RetroCraft**

Final Report

Members:

* Fernando Luo **(fbl2108)**
* Papoj Thamjaroenporn **(pt2277)**
* Lucy He **(lh2574)**
* Kevin Lin **(kl2495)**

**Table of Contents**

1. Introduction

2. Language Tutorial

3. Language Reference Manual

4. Project Plan

5. Architectural Design

6. Test Plan

7. Lesson Learned

8. Appendix

1. Introduction

1.1 Language Overview

RetroCraft is a programming language that aims to provide users with the tools to easily and creatively design a computer game. Our language focuses specifically on side-scrolling, obstacle-aversion style games. Games produced would be similar to Helicopter: a simple platform game in which the player has to keep a helicopter flying through a generated scene as far as possible without being hit by obstacles. Our language supports basic and more advanced functionalities including arithmetic operations, control flow, user-defined functions, recursion, and arrays of primitive types and scene objects. Combining built-in objects and functions that assist the coder’s creative process with imagination and intuitive code, our language is a powerful tool that casual gamers can easily use to generate their own game with impressive results.

Insert Helicopter Image here + Our Helicopter counter part!!!!

1.2 Background

Since the creation of platform games in the 1980s, video gamers have witnessed the growth and evolution of 2D platformers. The genre persists today with various legacies of games such as Super Mario Bros and Donkey Kong. However, gamers and hobbyists rarely have the chance to design their own. We have implemented a language that provides users with the building blocks to conveniently and creatively design their own game level, specifically for a game of a similar kind to Helicopter. RetroCraft defines an intuitive syntax that will allow the programmer to express the boundaries of a level, scene generation mechanics, and player characteristics. The language also provides powerful built-in functions that will execute game mechanisms without any specification from the user. These features include: collisions detection of generalized polygons, infinite loops that update the scene, the image generation mechanism, the score of the player, and the input events that detect keyboard input and respond accordingly automatically.

2. Language Tutorial

2.1 File Extension

Our language executes source code with “.rc” extension.

2.2 Compiling and Running Test Cases

Our language comes with a Makefile that can be used to easily compile our language compiler. To run the source code, execute:

./retrocraft [options] < [.rc files]

Options:

-b Generate the byte code

-c Compile the source code (default)

2.3 Generating Test Cases Reference

We provide a shell script testall.sh which can be executed to either: generating test case references, or running the testing source codes in the test suite against the references. The command is the following:

./testall.sh [options] [.rc files]

Options:

-k Keep intermediate files

-r Generate test references instead of running code against them

-h Print this help

If the file is not specifies, the script will run the code through all source codes that live within the main directory. Please note that to be able to test the codes or generate the references, one must make first

2.4 A Simple Program: Greatest Common Divisor

The following program evaluates the greatest common divisor of a given set of three integer pairs. Through this sample, we demonstrate the concept of user-defined functions, function calls, and flow control (an if statement and a while loop).

function $gcd : int (int $a, int $b)

{

while ($a != $b)

{

if ($a > $b)

$a -: $b;

else

$b -: $a;

}

return $a;

}

function $main : void ()

{

$printstring("Should print 2, 3, and 11");

$printint( $gcd(2,14) );

$printint( $gcd(3,15) );

$printint( $gcd(99,121) );

}

2.5 A Simple Helicopter Game

Insert our sample code for Helicopter HERE!!!!!

For a more sophisticated sample code please refer to the

2.6 A More Complex Sample: Generating the Obstacles

Insert our sample code for Helicopter HERE!!!!!

3. Language Reference Manual

**3.1 Lexical Convention**

**3.1.1 Comments**

Double forward slashes // indicate the beginning of a single line comment. Multiple line comments will begin with /\*and end with \*/.

**3.1.2 Tokens**

The types of tokens in our language are: keywords, identifiers, constants, string literals, operators and separators.

**3.1.2.1 Keywords**

RetroCraft has a list of reserved words with fixed purposes.

Variable type declaration: int, string, function, void

Control flow: if, else, while, for, return, true (1), false (0)

Data object: Array, Map, Player, Brick

**3.1.2.2 Identifiers**

Identifiers begin with a dollar sign ( $ ) followed by a sequence of upper and/or lowercase characters, digits and underscores, starting with a non-numerical character. The keywords in 2.2.1 are not valid identifiers. Upper and lower case characters are unique, making identifiers case-sensitive.

**3.1.2.3 Separators**

|  |  |
| --- | --- |
| \t | tab |
| \n | new line feed |
| \r | return |
| <space> | space |

**3.1.2.4 Punctuators**

|  |  |
| --- | --- |
| ; | end of line |
| , | separates arguments, object attributes |
| { } | code block |
| “ … ” | double quotes for string |
| () | function calls or arithmetic operations |
| [ ] | array random access |
| . | referencing object’s attributes and functions |

**3.1.3 Operators**

**3.1.3.1 Arithmetic**

Our arithmetic operators will be the standard operators present in most languages. The symbols and associated operations are as follows:

|  |  |
| --- | --- |
| : | Assignment |
| +,- | Addition and Subtraction |
| +:,-:, \*:, /: | Shorthanded Add, Subtract, Multiply, and Divide |
| \*,/ | Multiplication and Division |
| % | Modular |

Arithmetic expressions will be made using infix notation, i.e. operand1 operator operand2. The standard order of operations specified by arithmetic will be honored, i.e. PEMDAS. Arithmetic operates on type int.

**3.1.3.2 Comparison**

|  |  |
| --- | --- |
| = | Equal |
| != | Not equal |
| > | Greater than |
| < | Less than |
| >= | Greater than or equal |
| <= | Less than or equal |

These operators compare variables and/or constants with each other and return an integer constant ( 1 for true, and 0 for false). Incompatible types will result in a syntax error.

**3.1.3.3 Logical Operators**

|  |  |
| --- | --- |
| && | AND |
| || | OR |
| ! | NOT |

Logical operators can be used with expressions which evaluate to either 1 or 0. The order of precedence is: NOT, then AND and OR. It is recommended that a parenthesis is used when an expression involves multiple logical operators, e.g., ($x = 3) || (($x = 4) && ($y = 1)) instead of ($x = 3) || ($x = 4) && ($y = 1)

**3.1.3.4 Member Operators**

Member operators on objects will use a single dot ( . ) notation. For example, to access the $height property of a Map object $gameMap, the notation $gameMap.$height should be used.

Member operators on our zero based arrays will use a square bracket notation. For example, to access the 2nd index of an array $sampleArray, the notation $sampleArray[1] should be used.

**3.2 Statements**

**3.2.1 if, else if, else**

if, else if and else statements are used to control when their contained blocks of code will be executed. For example:

if (<logical expression>) {

// code executed if above expression evaluated to true

} else if (logical expression) {

// code executed if first logical expression was false

// and the second was true

} else {

// code executed if both logical expressions were false

}

**3.2.2 for**

for statements are used to control the number of times a block of code is executed. The for statement has three components:

for (<variable initialization> ; <logical expression> ; <variable increment/decrement>) {

// code to execute

}

The code will continue to be executed as long as the logical expression is true. The variable initialization and increment/decrement give a compact way to control the number of times the code is executed. Code block following the for statement must be wrapped in brackets.

For example, the following would iterate through the code 5 times:

int $i;

for ($i : 0; $i < 5; $i +: 1) {

// code to execute

}

**3.2.3 while**

A while loop evaluates the bracketed statements if the given logical expression remains true.

while (<logical expression>) {

// code to execute

}

**3.2.4 return**

Functions terminate when they reach a return statement. If the function has a return type, return must be followed by a value of that type.

**3.3 Declarations and Assignments**

**3.3.1 Primitives**

RetroCraft supports two primitive types: int and string. We can declare a new primitive variable using the following syntax:

// Declaration and Assignment done separately

<primitive type> $<var\_name>;

$<var\_name> : <value>;

For example:

int $myInt;

$myInt : 5;

There is one thing we need to point out regarding the string type. According to how we designed the memory allocation, we have decided to allocate 40 words (1 word = 4 bytes) on the stack for a string. For this reason, the user will be able to use the string variable safely as long as the length of the string is not longer than 38 characters (the other two words are necessary for bookkeeping purposes on the stack). **Section 5** will discuss more about the architecture design.

**3.3.2 Arrays**

RetroCraft fully supports arrays of all types (int, string, Brick, Player, Map). Similarly to with primitives, an array must be declared first, and then initialized using the keywords new Array <type>. To access or define elements in an array, we use square brackets containing the desired element index. The syntaxes are shown below:

// Declaration, allocation, and assignment done separately

Array <object\_type> $<name\_of\_array>;

$<name\_of\_array> : new Array int;

$<name\_of\_array>[0] : <some\_data>;

$<name\_of\_array>[1] : <some\_data>;

...

For example:

Array int $arrayOfInts;

$arrayOfInts : new Array int;

$arrayOfInts[0] : 4;

$arrayOfInts[1] : 1;

$arrayOfInts[2] : 2;

The way we can access an array element is the following:

$arrayOfBlocks[1]

The index of any array starts from zero.

There are two aspects of the array we need to point out here. First, notice how the size of the array is never needs to be specified. We would like to simulate a dynamic array in our program. However, the actual array is always allocated 100 slots (i.e. for 100 elements, regardless of type). Therefore, the user can use the array as long as the number of elements does not exceed 100. Second, to initialize the array the keywords new Array are used to label pieces in memory as belonging to a certain type of array. In fact, new can also be used to create game objects including Map, Player, and Brick. Section 5 will explain more about this in details.

The size of elements in the array can also be accessed by the attribute length. For example:

$countArray()

**3.3.3 Function Declaration**

Function declarations begin with the keyword function. The header will also contain the return type and formal parameters. If there is no return type, void should be used instead.

function func\_name : <return type> (<parameters>) {

// Implementation

};

For example,

function $retMapArray : Array Map (int $total)

{

Array Map $retArray;

Map $m1;

int $i;

$retArray : new Array Map;

for ($i : 0; $i < $total; $i +: 1){

$m1 : new Map (768, 1024, $generateThis);

$retArray[$i] : $m1;

}

return $retArray;

}

We will inherit the same mechanism on parameter passes from OCaml: all parameters are implicitly passed by reference.

**3.4. Primitive Data Types and Basic Data Types**

Our language has five unique data types and another five data types which are just array types for the first five. These five unit types are outlined in the tables below.

**3.4.1 Primitive Data types**

|  |  |
| --- | --- |
| int | ..., -1, 0, 1, ... |
| string | “Hello World” |

**3.4.2 Basic Data types**

|  |  |  |  |
| --- | --- | --- | --- |
| Array  (See 3.3.2) | Stores a collection of data elements of the data type. Array elements are accessed with square brackets.  *Attributes*   |  |  | | --- | --- | | $length | The length of the array | |
| Map | The canvas for the game. It is the container for all objects including Brick and Player in the game. It also contains the $geneartor function pointer that invokes function to build all blocks.  *Variable and Object Attributes*   |  |  | | --- | --- | | int $width | Width of the game screen | | int $height | Height of the game screen |   *Function Attributes*   |  |  | | --- | --- | | $generateThis | Function pointer that returns an array of blocks | |
| Brick | Fundamental building blocks of the game environment. User provides parameters: (R, G, B, $verticesArray, x, y).  *Variable and Object Attributes*   |  |  | | --- | --- | | int $colorR  int $colorG  int $colorB | User provided RGB values of the brick. | | int $x,  Int $y | x and y coordinates of the block object | | Array int $verticesArray | pointer to an array of integers (vertices array) |   Brick objects will be translated along the map internally to simulate movement. Its movement is independent from Player. |
| Player | The user controlled character, which can be controlled to move through the map. Similar to Brick, user supplies the RGB values, pointer to the vertices array, and the starting Y position  *Variable and Object Attributes*   |  |  | | --- | --- | | int $colorR  int $colorG  int $colorB | User provided RGB values of the Player object. | | Array int $verticesArray | pointer to an array of integers (vertices array) | | int $y | y coordinates of the block object |   Player and Brick move independently of each other. User will be able to move the Player up and down (Y position). |

**3.5 Operations on Graphics Objects**

Since RetroCraft is primarily graphics based, we require a specific set of attributes and methods in order to control the layout and flow of the game. The following sections describe them.

**3.5.1 Object Construction**

Object variables are declared and constructed similar to the syntax specified in the variable declaration section above (3.3.1):

<object type> $<var\_name>;

$<var\_name> : <attributes>;

Instead of a primitive type, the variable name is preceded by an object type, specified as a data object keyword in section 3.1.2.1. Similar to the initialization of an Array, data object types uses the keyword new as well.

$myMap : new Map(700,500,$generate);

$b1 : new Brick(100,150,200,$vertices,20,30);

A detailed example:

function $main : void ()

{

Player $p1;

Array int $vertices;

$vertices : new Array int;

$vertices[0] : 400;

$vertices[1] : 200;

$vertices[2] : 150;

$vertices[3] : 300;

$p1 : new Player(0,0,255,$vertices,10);

}

To access the object and its attribute after creation, one can do a simple reference:

$printint($p1.$colorR);

$printint($p1.$colorG);

$printint($p1.$colorB);

$printint($p1.$y);

/\* player vertices \*/

$printint($p1.$vertices[0]);

$printint($p1.$vertices[1]);

$printint($p1.$vertices[2]);

$printint($p1.$vertices[3]);

**3.5.2 Display and Movement**

The game map is a grid of a user-determined height and width measured in pixels. Coordinates increment up and to the right, such that the bottom left space in the map has the coordinates (0,0). Game objects, such as players, enemy characters and walls, are rectangular shaped entities specified by height and width values and are placed on the game map grid at specified coordinates according to their $px and $py attributes. Upon rendering an object, the bottom left corner of the object is placed at the specified coordinate on the game map and the rest of the object spans the space above and to the right.

In order to simulate movement, we have provided an EventManager oracle which redraws the scene described by the game map and its objects at each timestep. The coordinate values of each object can be changed by any of the user defined functions assigned to its onKeyPressed, onUpdate, or onCollision attributes. For each frame, the EventManager oracle cycles through each of the objects on the map currently being run and calls the onKeyPressed function if a key is being pressed and updates each of the objects according to those functions. Then cycling through the objects a second time, the onUpdate function of each object is called to apply more changes. Finally, the EventManager cycles through all possible pairs of objects on the map to determine which pairs are at a point of collision, *a state we define as two objects whose bounding box perimeters are either in contact or overlapping*. Then for each of those objects found to be in a point of collision, their onCollision function is called with an input parameter of the object colliding with it in order to resolve those collisions.

Ideally, a user defined onKeyPressed function would be written to govern all changes to the object that user input would control, such as the increase of the velocity of the Player object when the user inputs a move forward key. Then the onUpdate function would be written to make changes to the object based on its current attributes and the passive rules of the environment, such as gravity and friction. Lastly, the onCollision function acts to apply the final checks to the system in the common case of object collision, such as making sure Player objects do not pass through the walls of the map.

For example, here is the definition of a player object on a map with wall objects on the south, east and west borders who starts on the accelerates to the right up to a certain speed as a user presses and holds down the ‘D’ key but gradually comes to a halt when the user lets go of the key. The reverse is also true if the user were to press and hold down the ‘A’ key. Additionally, when the player object runs into the wall object, it will come to an immediate halt.

PlayerObj $myPlayer: PlayerObj {

$height: 20.0,

$width: 10.0,

$px: 10.0, //assuming the walls and floor are 10px thick

$py: 10.0,

$vx: 0.0,

$vy: 0.0,

$playerImgs: Array Image {

Image {

$src: “images/playerImage.jpg”

}

},

$visible: true,

$onKeyPressed: void (Map $gameMap, char $keyPressed) {

if ($keyPressed = ‘d’) {

if ($vx >= 0.0) { $vx +: 2.0; }

else { $vx : 2.0; }

if ($vx > 10.0) { $vx : 10.0; }

}

else if ($keyPressed = ‘a’) {

if ($vx <= 0.0) { $vx -: 2.0; }

else { $vx : -2.0; }

if ($vx < -10.0) { $vx : -10.0; }

}

},

$onUpdate: void (Map $gameMap) {

// $timestep is a global variable of the game

$px : $px + $vx \* $timestep - 0.5 \* $gameMap.gx \*

$timeStep ^ 2.0;

},

$onCollision: void (Map $gameMap, Object $collidingObject) {

if (typeOf($collidingObject, EnvObject)) {

if ($px <= $collidingObject.px) {

$px : $collidingObject.px - $width;

$vx : 0.0;

}

else if ($px > $collidingObject.px) [

$px : $collidingObject.px + width;

$vx : 0.0;

}

}

}

};

**3.5.3 Modifying Objects**

Attributes of various objects can be modified after object creation by referencing the object ($<object name>) and using the punctuator ‘.’ to call attributes:

function $main : void ()

{

Array int $vertices;

Brick $b1;

$vertices : new Array int;

$vertices[0] : 567;

$vertices[1] : 420;

$b1 : new Brick(100,150,200,$vertices,20,30);

$b1.$colorR : 255;

$b1.$colorG : 255;

$b1.$colorB : 255;

$b1.$vertices[0] : 121;

$b1.$vertices[1] : 408;

$b1.$x : 0;

$b1.$y : 0;

}

**3.5.4 Advanced Attributes and Functions of Object’s**

The object does not only provide basic attributes such as width and height of the object, but also some functionality that, after being defined by the user, can be used to control the behavior of the object and its interaction with other objects.

**3.5.4.1 Dimensions**

Each object’s dimension attributes, $height and $width, define the rectangular area of pixels allotted to it on the grid.

**3.5.4.2 Coordinate Location**

Each object’s coordinate attributes, $x and $y. These coordinates could be changed over the course of a game with internal keyboard events.

**3.5.4.3 generateThis (Map)**

The Map object has a pointer to a function that generates and returns an array of Bricks. This function will be invoked as the game progresses to draw blocks. User can program it to dynamically change the map depending on the score.

**3.6. Built-in & Required Functions**

**3.6.1 main**

Every game created by RetroCraft requires a main function. All games will begin execution from this function.

The $main() function is composed of two main sections. The first section includes the initialization of all variables. The next section follows normal program flow; provide that any necessary initializations are done first.

**3.6.2 Run (Map $mapObject, Player $playerObject)**

The $Run function takes a Map and a Player object and invokes the helper built-in functions: $DrawPlayer and $CallGenerator. It builds the game with necessary bookkeeping functions and displays the game onto a graphics window.

**3.6.3 DrawPlayer (Player $playerObject)**

The $DrawPlayer function takes a player object and paints it on the graphics window.

**3.6.4 printint (int $i) or printint (1)**

Prints an integer literal or a integer variable onto the console. Retrocraft will type check the parameter to ensure that this function prints only data type int.

**3.6.5 printstring (string $str) or printstring(“hello”)**

Prints a string variable or a string literal with a maximum length of 38 characters. Retrocraft will type check the parameter to ensure that this function prints only data type string.

**3.6.6** **dumpstack()**

The dumpstack function allows user to display the entire stack on console. This allows for ease of debugging and for one to access and trace through the memory structure

**3.6.7** **CallGenerator (Map $mapObject)**

This function will invoke the $generator function inside the given Map object and create the block of Bricks necessary for display.

**3.6.8** **Push (Array <type> $in\_array, <type> $object)**

The push function will push an object into the specified array. If the array is full, an exception will be thrown

**3.6.9** **GetCurrentScore ()**

A built-in function that allows user to obtain the score within a lifetime of a game and put it on top of the stack.

**3.6.10 GenerateRandomInt (int $i)**

User can use this function to generate a random integer using another integer as a seed. Retrocraft will type check to make sure that the parameter is indeed an integer.

**4.** Project Plan

**5. Architecture Design**

**5.1 Parser/Scanner**

Inspired by the MicroC compiler, RetroCraft utilizes the Scanner in conjunction with the Parser to read the program and generate the abstract syntax tree of the program. The scanner file first converts the source code into discrete tokens. Rules in the scanner file allows for multiline and single line comments. We’ve also identified all the reserved keywords (**Section 3**) as tokens to prevent users from mistakenly use them as variables. Furthermore, our scanner guarantees that all identifiers start with ($).

The parser invokes the program routine to generate a list of variable declaration and a list of function declaration. This architecture satisfies our language due to the presence of global variables.

function $main : void () {

int $i;

$i : 0;

$printint ($i);

}

**would be translated to:**

FUNC ID ASSIGN type(“void”) LPAREN formals\_opt RPAREN LBRACE

INT ID SEMI

ID ASSIGN (expr-> LITERALINT) SEMI

ID LPARENT actuals\_opt RPAREN SEMI

RBRACE

**Next it would be parsed to:**

{ fname = “$main”;

formals = ();

locals = $i;

Body = Assign ($i, 0);

@ Call (“printint”, $i)

}

**Then finally into bytecode:**

0 OpenWin

1 Jsr 3

2 Hlt

3 Ent 2

4 Init 1 2 1

5 Litint 0

6 Sfp 2

7 Drp

8 Lfp 2

9 Jsr -3

10 Drp

11 Litint 0

12 Rts 0

**To be executed:**

$ ./retrocraft < test/test.rc

0

**5.1.2. AST**

The AST first enumerates the tokens and specify and associativity between operators to reflect standards such as PEMDAS. The abstract syntax tree primarily defines the core structure of a retrocraft program. The parser will reference this file in order to generate an tree.

**5.1.3. Bytecode**

Our bytecodes are as follows:

**Litint of int** (\* Push a int literal \*)

**Litstr of string** (\* Push a string literal \*)

**Drp**  (\* Discard a value \*)

**Bin of Ast.op**  (\* Perform arithmetic on top of stack \*)

**Lod of int**  (\* Fetch global variable \*)

**Str of int**  (\* Store global variable \*)

**Loda**  (\* Load global array variable \*)

**Stra**  (\* Stores global array variable \*)

**Lfp of int**  (\* Load frame pointer relative \*)

**Sfp of int**  (\* Store frame pointer relative \*)

**Lfpa**  (\* Index is evaluated and put on top of stack\*)

**Sfpa**  (\* Stores frame pointer of array \*)

**Lref**  (\* Loads a value onto the stack from an address \*)

**Sref** (\* Saves a value from the stack into an address \*)

**Jsr of int**  (\* Call function by absolute address \*)

**Ent of int**  (\* Push FP, FP -> SP, SP += i \*)

**Rts of int** (\* Restore FP, SP, consume formals, push result \*)

**Beq of int**  (\* Branch relative if top-of-stack is zero \*)

**Bne of int**  (\* Branch relative if top-of-stack is non-zero \*)

**Bra of int**  (\* Branch relative \*)

**Make of int** (\* Shift stack pointer by 1 for Player, Map, Brick; Adds vartype\_id to first space in arrays \*)

Init of int \* int \* int

**OpenWin** (\* Opens a display window \*)

**CloseWin** (\* Closes the display window \*)

**DrawPlayer** (\* Draws a player object on top of the stack \*)

**CheckCollision** (\* Checks if the player object has collided with anyone \*)

**CheckUserInput** (\* Checks for user input and modifies player on stack \*)

**PrintScore** (\* Prints the user's current score on the top left \*)

**Hlt** (\* Terminate \*)

**Nt** (\* Negate 1 or 0 on top of stack \*)

**5.1.4 Execute.ml**

Due to more types than MicroC, we needed to differentiate our stack values from each other with an int typeID. The execute will read the bytecode, allocate a stack, and perform stack operations based on the program. Execute.ml maintains stack, frame and program pointers. Execute is also responsible for opening graphics window and performing object translations graphically due to close proximity to the actual data.

**5.1.5 retrocraft.ml**

This is the command line program that allows user to output the bytecode of the program instead of compiling. It traces through each command, displaying any pertinent information regarding stack operations, which makes it ideal for debugging.

**6. Test Plan**

To demonstrate the power of our language, we created various test cases to see the limit of our language. Retrocraft can handle from basic arithmetic to even slight more complex math that employs recursion. (Fibonacci’s series).

Retrocraft has great support for (while, for) loops while endured numerous testing of ‘if’ and ‘else’ logic. Our language allows referencing of ids and also supports returning of all data types.

Array support is further tested by combining the tests with various data types and looping logic. Furthermore, we have included an automated testing script which will compare the output of each file to the supposed output of the test programs (testall.sh).

7. Lessons Learned

**8. Sample Code**

**Appendix 1:**

Test-array.rc

function $retIntArray : Array int ()

{

Array int $retArray;

int $i;

$retArray : new Array int;

for ($i :0; $i < 5; $i +: 1){

$retArray[$i] : $i;

}

return $retArray;

}

function $retIntArray2 : Array int ()

{

Array int $retArray;

int $i;

$retArray : new Array int;

for ($i :0; $i < 15; $i +: 1){

$retArray[$i] : $i;

}

return $retArray;

}

function $main : void ()

{

Array int $localArray;

int $i;

$localArray: $retIntArray();

$printstring("printing returned array");

for ($i : 0; $i < 5; $i +: 1) {

$printint($localArray[$i]);

}

$localArray: $retIntArray2();

$printstring("printing 2nd returned array");

for ($i : 0; $i < 15; $i +: 1) {

$printint($localArray[$i]);

}

}

**Appendix 2:**

**Test-brick.rc**

Array int $vertices;

Brick $b1;

Brick $b;

function $retBrickArray : Array Brick ()

{

Array Brick $retArray;

int $i;

int $j;

int $k;

$vertices : new Array int;

$vertices[0] : 1;

$vertices[1] : 2;

$vertices[2] : 3;

$vertices[3] : 4;

$j : -1;

$k : 0;

$retArray : new Array Brick;

for ($i : 0; $i < 20; $i +: 1){

if (($i % 5) = 0) {

$j \*: -1;

}

$k +: $j;

$b1 : new Brick (0,0,0, $vertices, $i, $k);

$retArray[$i] : $b1;

}

return $retArray;

}

function $main : void ()

{

Array Brick $brickArray;

int $i; int $total;

$total : 20;

$brickArray: $retBrickArray();

$printstring("printing returned array of bricks");

for ($i : 0; $i < $total; $i +: 1) {

$b : $brickArray[$i];

$printstring("Printing Block: ");

$printint($b.$colorR);

$printint($b.$colorG);

$printint($b.$colorB);

$printint($b.$vertices[0]);

$printint($b.$vertices[1]);

$printint($b.$vertices[2]);

$printint($b.$vertices[3]);

$printint($b.$x);

$printint($b.$y);

}

}

**Appendix 3:**

**Test-player.rc**

function $retPlayerArray : Array Player (int $total)

{

Array Player $retArray;

Array int $vertices;

Player $p1;

int $i;

$vertices : new Array int;

$vertices[0] : 0;

$vertices[1] : 0;

$vertices[2] : 20;

$vertices[3] : 20;

$retArray : new Array Player;

for ($i : 0; $i < $total; $i +: 1){

$p1 : new Player (255,255,255, $vertices, 0);

$retArray[$i] : $p1;

}

return $retArray;

}

function $main : void ()

{

Array Player $playerArray;

Player $p;

int $i; int $total;

$total : 30;

$playerArray: $retPlayerArray($total);

$printstring("printing returned array of bricks");

for ($i : 0; $i < $total; $i +: 1) {

$p : $playerArray[$i];

$printstring("");

$printint($i);

$printint($p.$colorR);

$printint($p.$colorG);

$printint($p.$colorB);

$printint($p.$y);

}

}

**Appendix 4:**

**Test-map.rc**

function $generateThis : Array Brick ()

{

Array Brick $retArray;

Array int $vertices;

Brick $b1;

int $i;

int $j;

int $k;

$vertices : new Array int;

$vertices[0] : 0;

$vertices[1] : 0;

$vertices[2] : 10;

$vertices[3] : 10;

$j : -1;

$k : 0;

$retArray : new Array Brick;

for ($i : 0; $i < 20; $i +: 1){

if (($i % 5) = 0) {

$j \*: -1;

}

$k +: $j;

$b1 : new Brick (0,0,0, $vertices, $i, $k);

$retArray[$i] : $b1;

}

return $retArray;

}

function $retMapArray : Array Map (int $total)

{

Array Map $retArray;

Map $m1;

int $i;

$retArray : new Array Map;

for ($i : 0; $i < $total; $i +: 1){

$m1 : new Map (768, 1024, $generateThis);

$retArray[$i] : $m1;

}

return $retArray;

}

function $main : void ()

{

Array Map $mapArray;

Map $m;

int $i; int $total;

$total : 3;

$mapArray: $retMapArray($total);

$printstring("printing returned array of maps");

for ($i : 0; $i < $total; $i +: 1) {

$m : $mapArray[$i];

$printstring("");

$printint($i);

$printint($m.$height);

$printint($m.$width);

}

}

**Appendix 5 Source Code:**