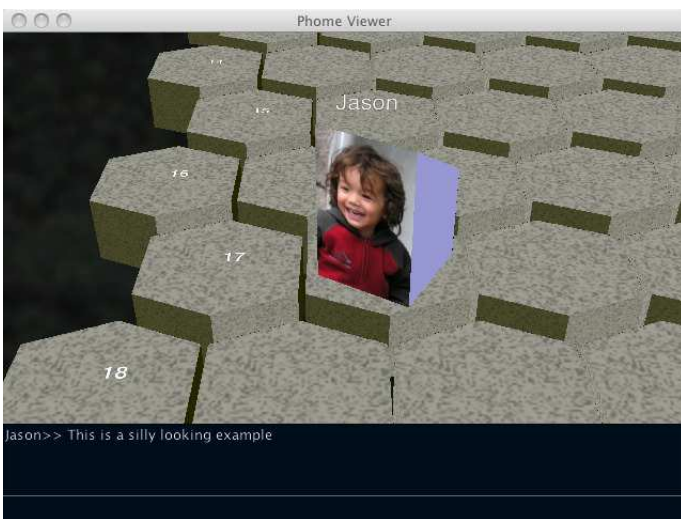


1 Ant II: Speaker for the Dead

For this lab, we'll descend one more step, moving even closer to the machine. This time, instead of compiling to the OCaml virtual machine, the objective is to implement the virtual machine itself. We'll be working in a networking environment, where you receive bytecode programs (in `.mls` format) over the network, and you are to evaluate them.

The bytecode programs can be potentially malicious, so you will need to ensure that your machine implementation is safe. Fortunately, bytecode programs are not allowed to access the filesystem.

The lab is based in a virtual world that uses the ant geometry from Lab 1. That is, there is a 2d hexagonal grid, where you can move around. This time, instead of collecting food, your objective is simply to participate in a shared virtual world, much like a MMORPG (but without killing one another).



2 Architecture

The project architecture is divided into four programs, shown in Figure 1. There is a centralized server `phome_server` for the virtual world. A client is constructed as a pipeline; the `phome_client` reads commands from the server, passes them to the virtual machine, which displays its output in the viewer `PhomeViewerApp`. The viewer also sends keyboard commands to the server.

For this lab, your task is to implement the virtual machine. Everything else is provided. You don't have to deal with networking, and you don't need to know very much about graphics, but you do have to know how to load a program from a file and execute it.

The virtual machine is designed as a filter. It reads commands from the standard input channel, executes them, and writes the results to the standard output channel. Each of these channels has its own line-oriented command format.

3 Client commands

The client produces the following commands. Each line forms a command where the first character identifies the command, and the remaining text represent the arguments.

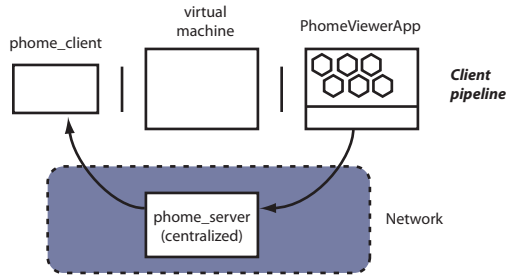


Figure 1: The four Phome applications.

P name:string filename:string
Load a program <code>name</code> from the file <code>filename</code> . Evaluate the expression <code>(name ("initialize"))</code> .
K name:string
Kill the program <code>name</code> and remove it from the virtual machine.
C name:string command:string args:string list
Call the program <code>name</code> with the given string arguments. That is, evaluate <code>(name '(command args))</code> .
T now:float
Call all programs with the given “timeout”. That is, for each program <code>name</code> , evaluate <code>(name '("timeout" now))</code> .
L name:string x:int y:int
The avatar for <code>name</code> has moved to location (x, y) . For each program <code>name</code> , evaluate <code>(name '("location" x y))</code> .

4 Viewer

The viewer uses the following definitions and specification.

- An *avatar* represents the image of a character in the virtual world. Each avatar is drawn in a cube based on the origin with the bounds $-0.5 \leq x \leq 0.5$, $-0.5 \leq z \leq 0.5$, $0 \leq y \leq 1$.
- A *panel* is a collection of triangles that forms the image of the avatar.
- A *triangle* is described by three vertexes.
- A panel has an *appearance* that describes its material, transparency, and texture.
- A *material* defines color properties of a surface.
- A *vertex* is a three-dimensional point (x, y, z) .

4.1 Viewer commands

The viewer takes the following commands. *Note:* there is no relationship between the character the specifies a command and the command itself.

4.1.1 Avatars

N name:string
Set the current avatar to <code>name</code> .

K name:string
Remove the avatar with the given name. The current avatar is unaffected.

4.1.2 Panels

P i:int
Set the current panel to <i>i</i> . Each avatar can have up to 100 panels.

R vertex_count:int
Reset the current panel, and allocate space for <code>vertex_count</code> vertexes. Each sequence of 3 vertices defines a triangle. The <code>vertex_count</code> must be a multiple of 3. The coordinates of the vertex must be specified before the panel can be drawn.

D
Display the current panel.

X scale:float
Apply a uniform scale to the current panel. All transformations can be applied while the panel is displayed.

Y dx:float dy:float dz:float
Apply a translation to the current panel.

Z x:float y:float z:float w:float
Apply a rotation to the current panel.

c i:int r:float g:float b:float
Set the color for vertex <i>i</i> in the current panel.

x i:int x:float y:float z:float
Set the coordinates for vertex <i>i</i> .

A i:int
Set the appearance for the current panel to appearance <i>i</i> .

4.1.3 Materials

m i:int
Set the current material to <i>i</i> . Each avatar can have up to 100 materials. Material <i>i</i> is also reset by this command.

b r:float g:float b:float
Set the ambient color of the material.

e r:float g:float b:float
Set the emissive color (the glow) of the material.

d r:float g:float b:float
Set the diffuse color of the material.

s r:float g:float b:float
Set the specular color of the material.

y amount:float
Set the shininess of the material.

4.1.4 Appearances

a i:int
Set the current appearance to <i>i</i> . Each avatar can have up to 100 appearances. The appearance is reset by this command.

v amount:float
Set the transparency of the appearance.

k i:int
Set the material used by this appearance.

l url:string
Set the appearance's texture. The <i>url</i> should be a URL that points to a square image. The size of the image should be a power of 2, and no more than 256.

A texture is an image that is overlaid onto a surface. If you are going to use textures, you have to specify texture coordinates for each vertex.

t i:int x:float y:float
Set the texture coordinates for vertex <i>i</i> . The texture is viewed as a 2D surface, where (0,0) is the lower left, and (1,1) is the super right.

4.1.5 Other

M from:string to:string text:string
Send a text message from avataer <i>from</i> to avatar <i>to</i> . The format of the text is unspecified—you choose the format.

G dir:string
Move your avatar in one of four ways.
forward Move forward.
back Move backward.
left Rotate left.
right Rotate right.

S from:string text:string
Say something. The message is displayed on the screen, but not passed to the avatar programs.

w duration:float
Pause the viewer some number of seconds. This is mainly for debugging.

5 The Virtual Machine

Each avatar is defined by a bytecode program. The sequence is as follows.

1. You write a bytecode program to display your avitar.
2. The program broadcast to all the clients.
3. Each client executes the program, which draws the image of the avatar, optionally defines some animation, and reacts to various events.

The bytecode is based on the OCaml virtual machine, with some extra instructions for the graphics operations. The machine also supports floating-point values.

The instruction set is the same as for Lab 3, with the following exceptions.

- The `push_retaddr` instruction is *always* required.
- The instruction `apply` works with any number of arguments.

5.1 Core instruction set

<code>acc i</code>	Set the accumulator to stack element i .
<code>accu \leftarrow sp[i]</code>	
<code>envacc i</code>	Set the accumulator.
<code>accu \leftarrow env[i]</code>	
<code>const c</code>	Set the accumulator to a constant value. The value c can be a string, integer, or floating-point literal.
<code>accu \leftarrow c</code>	
<code>push</code>	Duplicate the accumulator. The accumulator is unchanged.
<code>*sp- \leftarrow accu</code>	
<code>pop i</code>	Pop n elements of the stack, not including the accumulator.
<code>sp += n</code>	
<code>assign i</code>	Set an element of the stack to the accumulator.
<code>sp[i] \leftarrow accu</code>	
<code>apply</code>	The accumulator should contain a closure, and the stack holds the arguments to the function being called. The function in the closure is called, returning a value in the accumulator, and removing the arguments from the stack.
<code>grab n, restart</code>	These are now no-ops. We do not allow partial application.

push_retaddr label

For **all** function applications, you must construct the calling frame manually. The sequence is: a) push the return address b) push the arguments to the function, c) use **apply**, d) label the code after the application. For example, suppose the accumulator contains a closure that expects 4 arguments. Here is a code sequence to apply the function then the number 10, 11, 12, and 13.

```
// Assumes sp[0] contains the closure
push_retaddr L1      // Push the return address (stack grows by 2)
const 13
push
const 12
push
const 11
push
const 10
push
acc 6                // Get the closure
apply 4              // Apply the function
```

L1:

return n

Return from a function, removing n values from the stack (not including the accumulator).

closure label, n

Create a closure, where the function body is at the location denoted by *label*, and the size of the environment has n elements. The stack (including the accumulator) should contain the values for the environment. These values are removed from the stack, and the accumulator is replaced with the closure.

closurerec label, n

Not allowed.

offsetclosure n

Set the accumulator to the current closure, offset by n . The only allowed value of **n** is 0.

accu \leftarrow env + n

setglobal identifier

Store the accumulator in the global value.

global-identifier \leftarrow accu

getglobal identifier

Replace the accumulator with the value stored in the global variable.

accu \leftarrow global-identifier

makeblock n, m

Allocate a memory block containing n values, with tag m . The values are the top n elements of the stack (including the accumulator). The values are removed, and the accumulator is replaced with the returned value. The tag m is ignored.

getfield n

The accumulator should point to a block. Replace the accumulator with the n^{th} field. There is no safety checking; the result is undefined if the accumulator does not contain a block, or if the index is out of bounds.

setfield n	The accumulator should point to a block, and the next element on the stack is the value to be stored in the block. Set the n^{th} field of the block to the value, and remove the value from the stack. There is no safety checking; the result is undefined if the accumulator does not contain a block, or if the index is out of bounds.
	Field(accu, n) \leftarrow *sp++
vectlength	The accumulator should point to a block. Replace the accumulator with the number of words in the block.
	accu \leftarrow Length(accu)
branch label	Got the label.
	pc \leftarrow label
branchif label	If the accumulator is nonzero, goto the label. The accumulator is not affected.
	if accu \neq 0 then pc \leftarrow label
branchifnot label	If the accumulator is zero, goto the label. The accumulator is not affected.
	if accu = 0 then pc \leftarrow label
boolnot	If the accumulator is zero, replace it with one, otherwise replace it with zero.
	if accu = 0 then accu \leftarrow 1 else accu \leftarrow 0
neg	Negate the accumulator. The accumulator must hold an integer or floating-point value.
	accu \leftarrow -accu
add, sub, mul, div, modint, andint, orint, xorint, lslint, lsrint, asrint	Arithmetic on the top two elements of the stack. The first four instructions operate on integer and floating-point values. The accumulator is set to the result, and the second value is removed from the stack.
	accu \leftarrow accu <i>op</i> *sp++
eqint, neqint, ltint, leint, geint	Not allowed. Use the <code>cmp</code> instruction instead.

cmp cc	Comparisons. The accumulator is set to the Boolean result, represented with the integers 0 (for false) and 1 (for true). The condition code <i>cc</i> is one of eq , neq , lt , le , gt , or ge . The operands can be integers, floating-point values, mixed integer/floating-point comparisons, or strings.
if accu <i>op</i> *sp++ then accu ← 1 else accu ← 0	
isint, isfloat, isstring, isblock	Test whether the accumulator contains a value of the specified type. The accumulator gets the result.
if accu <i>is an block</i> then accu ← 1 else accu ← 0	
stop	Halt execution.

5.1.1 New instructions

memglobal identifier	Test whether the global identifier is defined.
if identifier is defined then accu ← 1 else accu ← 0	
setvectitem	Set a field of a block. accu is the block, sp[0] is the index, and sp[2] is the value.
accu[sp[0]] ← sp[1] sp += 2	
getvectitem	Get a field of a block. accu is the block, sp[0] is the index;
accu ← accu[sp[0]] sp++	
setstringchar	Set a character in a string. accu is the string, sp[0] is the index, and sp[2] is the character (represented as an ASCII integer).
accu[sp[0]] ← sp[1] sp += 2	
getstringchar	Get a character in a string. accu is the string, sp[0] is the index;
accu ← accu[sp[0]] sp++	

sin, cos, tan, sqrt
Standard trigonometric functions.
accu ← op(accu)
string_cat
Concatenate two strings.
accu ← accu sp[0] sp++
string_split
Split a string into a list of words. The words are separated by whitespace. Quotation symbols are not special. A list is defined using the Lisp representation. Each cell in the list is a block of length 2. The first element is an item in the list. The second element points to the next cell in the list. The integer 0 is used to terminate the list.
accu ← split(accu)
hexify
Compute the hex representation of a string. Each character in the original string is represented to two hex characters in the result. For example, the string "ABC" is equivalent to "\x41\x42\x43", so the hex representation is "414243".
accu ← hexify(accu)
unhexify
Invert the hexify operation. If the string is not in hex, the result is undefined.
accu ← unhexify(accu)
string_create
Create a new string of length accu. The contents of the string is unspecified.
accu ← new char[accu]
int_of_string, string_of_int, float_of_string, string_of_float
Perform the specified conversion.
accu ← op(accu)
print
Print the accumulator. This can be useful for debugging, but it has no real effect on the virtual machine. You can print in any way you wish (or not print at all).
print(accu)
debug "string"
Print a constant string. This is also for debugging. This form leaves the virtual machine state unchanged, and it doesn't require an argument in the accumulator.
print("string")

5.1.2 Graphics instructions

To keep it short, we list the instructions in abbreviated format, where the arguments are listed next to the instruction. The arguments are *not* part of the instructions; they are on the stack. The arguments are removed from the stack when the instruction is executed. The result in the accumulator is unspecified.

For example, addition would be written as follows.

add [i:int j:int]
accu ← i + j
This means, i is in the accumulator, j is sp[0], and the stack pointer is incremented by 1.
panel [i:int]
Print P i.
reset [size:int]
Print R size.
display
Print D.
scale [amount:float]
Print X amount.
translate [dx:float dy:float dz:float]
Print Y dx dy dz.
rotate [x:float y:float z:float w:float]
Print Z x y z w.
set_color [i:int r:float g:float b:float]
Print c i r g b.
set_coord [i:int x:float y:float z:float]
Print x i x y z.
set_texture_coord [i:int x:float y:float]
Print t i x y.
set_appearance [i:int]
Print A i.
material [i:int]
Print m i.
set_ambient_color [r:float g:float b:float]
Print b r g b.
set_emissive_color [r:float g:float b:float]
Print e r g b.
set_diffuse_color [r:float g:float b:float]
Print d r g b.
set_specular_color [r:float g:float b:float]
Print s r g b.
set_shininess [x:float]
Print y x.
appearance [i:int]
Print a i.
set_transparency [amount:float]
Print v amount.

set_texture [url:string]
Print 1 url.
set_material [i:int]
Print k i.
send_message [to:string text:string]
Print M <name> to text, where <name> is the name of the current program.
move [dir:string]
Print G dir.

6 Constraints

- An avatar program is expected to terminate within 1000000 time steps (and usually much less) each time it is run.
- The program is executed in an initial state where:
 1. The stack is empty.
 2. The accumulator contains the argument.
 3. The initial program counter is 0.
 4. The global variables are preserved across invocations.
- All illegal operations, runtime errors, *etc.* can result in program termination.

7 Other

Each triangle has a *normal* that is used for lighting. The normal is right handed, meaning that it points out of the triangle when the triangle is drawn in the counter-clockwise direction. Triangles look best looking into the normal.

8 What to do

You need to do the following:

- Implement a virtual machine.
- Implement an avatar program.

You should submit your project to subversion, in your individual account.

You should submit all your source code, as well as a README file containing a description of your project, as well as a qualitative description of your program's structure and how well-suited it was to the project, and any other comments you feel are important.

Your submission should include any programs written in higher-level languages (like Lisp) to construct the avatar bytecode program.

8.1 Grading

This lab is graded out of 100 points (as usual). There is no specific objective for your avatar program, except that it should display itself properly.

Part	Weight
Virtual machine	70
Avatar (correctness)	30

Extra credit	Weight
Judge's prize	20
Animation and AI	10
Beauty	10

The judge's prize is for the best avatar language designs (in the minds of the judges).

9 Getting started

This is a group lab. Make sure that you save enough time to write your avatars (don't spend it all on your virtual machine).

If you don't know graphics, ignore all the extra stuff like appearances, materials, transformations, *etc.*, at least in the beginning. You can start with just setting the colors of the vertexes. Appearances are nice, but correctness is more important.

Triangles are tedious to draw with. Consider writing a graphics library with other geometric items. Cubes, cones, spheres, *etc.*

The virtual machine must behave properly even for malicious avatar programs. If a program performs an illegal operation, you should terminate it.