DV1564

Interpreters and Scripting

2017 June 4

Project: Static Procedural Generation

Dr Andrew Moss

THIS IS A DRAFT - IT WILL CHANGE UP UNTIL WEDNESDAY'S LECTURE.

Purpose

The assignment is designed to assess your ability to embed a scripting engine inside a host application. The lower grades on the assignment test your understanding of the Lua C API by using it to implement custom C routines that affect Irrlicht from a Lua console. Higher grades demonstrate the design and implementation of a parser to handle a custom scene format. The highest grades are awarded for embedding fragments of Lua in the scene description that can manipulate and transform geometry on the way to Irrlicht.

Background

Each of the individual object descriptions corrosponds to a structure that should built in the Irrlicht data representation. There is an exact mapping between the data-description and the Lua data model, and the Irrlicht data model. These different representations of the data must be kep synchronised: when the Lua interpreter queries the data-model it must /reflect/ the current data-structures in Irrlicht. That is, the implementation of the API must query the current Irrlicht state and translate it to a Lua representation.

Submission and Assessment

The project must be archived into a gzipped tarball. Submission details will follow.

The final version of the code must be submitted by the deadline. Assessment will be an oral examination the week after the deadline. This will involve running the submitted version of the code, checking the criteria and answering some questions about the implementation.

Grading Criteria

Each of the grades is cumulative: the criteria for the lower grades must be met *and* the new criteria for the grade. i.e. to receive a grade C you must meet the criteria listed for grades E, D and C.

- Integrate Lua and Irrlicht as shown in the labs.
- Add the Irrlicht FPS controls so that they are active only when the window is in focus (i.e. if you switch to the terminal the FPS camera should stop grabbing the mouse and keyboard).
- Switch off back-face culling so that you do not need to respect the triangle winding order.
- Implement the addMesh, addBox, getNodes, camera and snapshot interfaces described below.
- As scene nodes are added using these interfaces names must be generated if not supplied, and unique IDs.

- The getNodes call return the type, name and id of each scene node in a table.
- Each of the interfaces must verify their arguments and generate appropriate errors for the test data given later.

> Grade E
> Demonstrate understanding of the Lua C API by fulfilling all the criteria above.

- Implement the addTexture, and bind calls as described below.

> Grade D
> Demonstrate understanding of integration issues by fulfilling all the criteria above.

- Define the scene configuration language from the example data given below.
- Implement the loadScene interface so that it destroys any current scene and instantiates the scene described in the given file.
- The `Lua(< ... >)` construct is not necessary for this grade.

> Grade C
> Demonstrate understanding of parsing by fulfilling all the criteria above.

- Implement the `Lua(< ... >)` construct so that meshes can be generated from code.
- Must execute the test cases given below.

> Grade B
> Demonstrate understanding of embedding issues by fulfilling all the criteria above.

- Implement a scene configuration with procedurally generated meshes contained in `Lua(< ... >)` mesh.
- Implement the `Lua(< ... >)` construct for textures, and to map code over the meshes in a Transform construct in the scene configuration.
- The scene should include procedurally generated textures.
- Demonstrate the use of the transform syntax in the scene to create multiple (unique) versions of the objects.
- Some ideas for suitable scenes: sierpinski cube with noise textures, simple L-system to create a tree...

> Grade A
> Demonstrate use of Lua for static procedural generation by fulfilling all the criteria.

Interface Description

Each of the following interfaces must be implemented in C, using the Lua C API, and linked into the lua_State so that they are available for use in the Lua interpreter running in the terminal.

> addMesh(vertex list)
> Create a scene node and add a custom mesh to it. The function takes one argument, a table containing vertices for a triangle list. Each vertex is a table with three numbers inside it. Error checking must be implemented to make sure that the argument is a valid list of triangle vertices. Errors should be shown in

the terminal.

```
addMesh({{0,0,0}, {5,0,0}, {0,0,5}})    -- Simple triangle in the plane
addMesh({"orange"},{},{})               -- Error: non-numeric coordinates
addMesh({{1,2,3,4},{},{})               -- Error: number of components
addMesh({{1,2,3})                       -- Error: not a valid number of vertices
```

addBox(position, size, [name])
Create a cube scene node with the given size, at the given position. The name is optional, if one is not supplied a name should be generated automatically. The function should check that the first argument is a valid 3d coordinate and the second argument is a number. If the third argument is supplied it must be a string.

```
addBox({0,0,0},1,"origin")      -- Cube of size one at the origin
addBox({1,0,0},1,"offx")
addBox({0,0,1},1,"offz")
addBox({0,0,2},1)               -- Name generated for node
```

getNodes()
Function returns a table with information about the current nodes in the scene. Each entry in the table shows the ID, name and type of node. An example of use is shown below:

```
for i=1,0 do addBox({i,i,i},1.0-i/10.0) end
for k,v in pairs(getNodes()) do for kk,vv in pairs(v) do print(k,kk,vv) end end
```

A sample of the output looks like this:

```
1       name
1       id      -1.0
2       name    name1
2       id      1.0
3       name    name2
3       id      2.0
...
```

camera()
Function moves the Irrlicht camera to the given position and points it at the given target coordinates. The implementation must check that both arguments are valid 3d coordinates.

```
camera({0,5,-20}, {0,0,0})
camera({0,5,-20}, {0,0})            -- Error: target has wrong number of coordinates
camera({0,5,-20}, {0,0,"x"})        -- Error: coordinate is not a number
```

snapshot(filename)

Function creates an image and writes it out the file specified. The function must take care of error checking so that if an I/O error occurs it is caught and reported in the terminal.

```
snapshot("output.png")
snapshot("///")                    -- Error: file could not be opened
```

addTexture(colour data, name)
The first argument to this function contains the colour data that is used to create a new texture in the driver. There are three levels in this table: rows, columns and RGB triples. The RGB components are passed as normalised values: Irrlicht cannot guarantee the texture format created as the decision is made by the driver. When this data is unpacked into the newly created texture it must be transformed onto the target colour space that the driver picked (e.g. A8R8G8B8 or A1R5G5B5).

This function must check that the table describes a square texture whose dimension is a power of two (e.g. 2x2, 4x4...). The name supplied must be used in the *irr::video::IVideoDriver::addTexture* call so that the bind operation can find it later and use it in a node's material.

```
addTexture({{{1.0,0.5,0.5},{0.5,1.0,0.5}},{{0.5,0.5,1.0},{0.0,0.0,0.0}}}, "yoyo")
-- No visible effect in scene - but should insert the texture into the driver.
```

bind(node name, texture name)
This function takes two arguments: the name of a scene node and the name of a texture. The scene node must be located in the scene graph by name, and its material updated to use the named texture. Any errors in the process must be caught and reported in the terminal.

```
addTexture({{{1.0,0.5,0.5},{0.5,1.0,0.5}},{{0.5,0.5,1.0},{0.0,0.0,0.0}}}, "yoyo")
bind("offz","yoyo")            -- Update the material for node offz to use the texture
bind("offz","something else")  -- Error: unknown texture
bind("dayz","yoyo")            -- Error: unknown node
```

loadScene(filename)
Load the scene configuration file from the filename specified, parse it into a tree structure, walk through the tree and build the resources specified, inserting them into the scene graph in Irrlicht so that they are visible. Error checking in this function should trap problems such as the file not existing and report Lua error messages.

```
loadScene("scene1.txt")
loadScene("file doesn't exist")        -- Should display an error message.
loadScene("badscene.txt")              -- If the file cannot be parsed display an error.
```

Implementation Notes

There will be hints and starting points here.

During development you will frequently want to rebuild your code and then run a particular test sequence against it in the terminal. Retyping the same series of commands into the interpreter gets tedious without an edit-buffer / history. One simple way around this is to insert the sequence that you want from the command

line, and then allow normal interaction with the program streams:

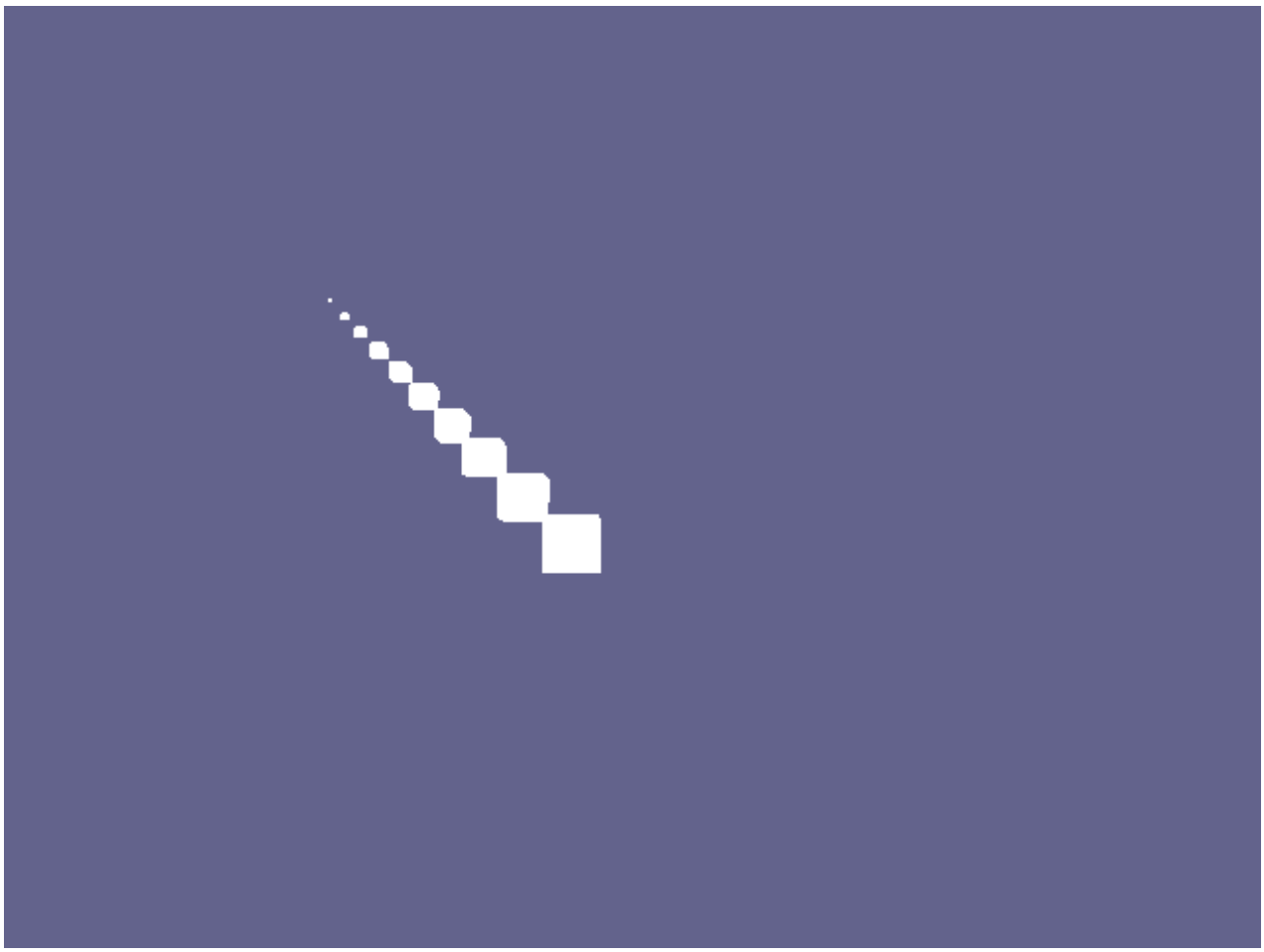
```
make project && (echo 'dofile("test2.lua")';cat) | ./proj
```

Test Data

test1.lua:

```
for i=1,10 do addBox({i,i,i},1.0-i/11.0) end
for k,v in pairs(getNodes()) do for kk,vv in pairs(v) do print(k,kk,vv) end end
camera({-10,0,0},{0,2,0})
snapshot("test1.png")
```

Lots more cases....



The scene configuration file contains a series of declarations of mesh data and textures. Each of these declarations defines the name that will be used internally to refer to that data. The Scene descriptor then places these resources into the actual scene. i.e. the declarations themselves will not create visible scene graph nodes, but each use of them inside the Scene will.

A simple scene description (scene1.txt):

```
Mesh("tri")
{
```

```
  (-20,0,-20),
  (20,0,-20),
  (-20,0,20),
}
Scene()
{
  Mesh("tri")
}
```

Loading the scene graph (test2.lua):

```
loadScene("scene1.txt")
camera({0,20,0},{0,0,0})
snapshot("test2.png")
```

A slightly more complex scene description (scene2.txt):

```
Mesh("square")
{
  (-20,0,-20),
  (20,0,-20),
  (-20,0,20),
  (20,0,-20),
  (20,0,20),
  (-20,0,20)
}
Texture("checkerboard")
{
  (0,0,0),
  (255,255,255),
  (255,255,255),
  (0,0,0)
}

Scene() { Bind("square", "checkerboard") }
```

A quite complex scene description (scene3.txt):

```
Mesh("bumpycyc")
Lua(<
  -- Build a planar height map
  radius = 5
  bumps = {}
```

```
  for row=0,9 do
    for col=0,9 do
      bumps[row*10+col] = math.random()
    end
  end

  -- Wrap the heightmap into a cylinder
  -- Transform from sparse 2d map into dense rows of verts
  cycRows = {}
  for row=0,9 do
    vrow = {}
    for col=0,9 do
      alpha = 2*math.pi * col / 10
      x = math.cos(alpha) * (radius-0.5 + bumps[row*10+col])
      z = math.sin(alpha) * (radius-0.5 + bumps[row*10+col])
      vrow[col] = {x,row*2,z}   -- Force from-0 for modulo below
    end
    table.insert(cycRows, vrow)
  end

  -- Pull out triangles from the vertex lists
  triVerts = {}
  for row=1,9 do
    rowBot, rowTop = cycRows[row], cycRows[row+1]
    for col=0,9 do
      bl, br, tl, tr = rowBot[col], rowBot[(col+1)%10], rowTop[col], rowTop[(col+1)%10]
      table.insert(triVerts,  bl)
      table.insert(triVerts,  br)
      table.insert(triVerts,  tl)
      table.insert(triVerts,  tr)
      table.insert(triVerts,  tl)
      table.insert(triVerts,  br)
    end
  end
  return triVerts
>)


Mesh("base"){
  (-20,0,-20),
  (20,0,-20),
  (-20,0,20),
```

```
  (20,0,-20),
  (20,0,20),
  (-20,0,20)
}

Texture("checkerboard")
{
  (0,0,0),
  (255,255,255),
  (255,255,255),
  (0,0,0)
}
Scene()
{
  Mesh("bumpycyc")
  Transform( "move10", Mesh("bumpycyc") )
  Bind("base", "checkerboard")
}
```