Blender's Game Engine as a 3D Environment Simulator for External Programs

Albert Cardona [ab] *

[a] Molecular Cell Developmental Biology, University of California Los Angeles, 621 Charles E. Young Dr. South, 90015 CA.
[b] Institute of Neuroinformatics, University of Zurich / ETHZ. Winterthurerstrasse 190, CH-8057 Zurich, Switzerland.

1 INTRODUCTION

The Monster Truck (nickname for a customized Traxxas E-MAXX) is a remote-controlled miniature 4-wheel drive car, equipped with suspension, strong grip wheels and shocks, and a USB input port for its steering and throttle systems. Toby Delbruck built onto the truck the necessary components to enable autonomous driving following visual cues. Namely:

- a neuromorphic silicon retina over the front end of the car, inside a protective hard-wood box containing a mirror to point it towards forward. Each light sensor in the retina's grid fires asynchronously and an event occurs on trespassing a certain threshold of contrast difference with the previous state.
- the jAER software package, which is a Java program for capturing, sequencing, viewing and especially processing address-event representation data, a protocol used for communication among silicon neurons. The software processes information packets (events) asynchronously, emulating the way in which neurons transmit and process information.
- a driver extension for the jAER, which processes visual input and extracts visual cues such as lines, computes the desired steering direction and speed, and feeds them back to the truck.
- a small Sony Vaio running the jAER.

For the fun of it, and because we could, we build a truck simulator in Blender. First we made a 3D mesh representing the truck, and then setup the game logic and a set of accessory python scripts to provide input to the jAER via network sockets.

The 3D simulator provides a totally editable arena for testing the trucks software. In particular, we are interested in evaluating the performance of the line extractor and the truck driver systems in the presence of heavily textured backgrounds.

The target is to make the truck drive autonomously following a painted line on the ground.

2 THE SIMULATOR COMPONENTS

There are four main elements in the simulator:

- The **game arena**: provides a solid ground (an actor with *dynamic* disabled) to run the car and to place textures and obstacle objects and ramps.

*to whom correspondence should be addressed: cardona@ucla.edu

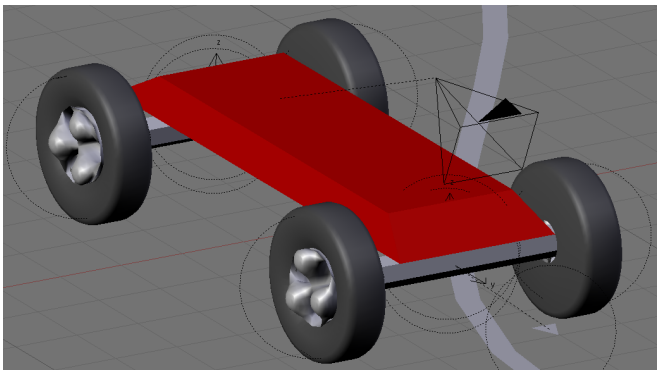

Fig. 1. A simple truck, with chassis and attached camera, and the arrow object at bottom right.

- The **truck**: the visible actor, composed of a tilting chassis over four wheels, the front two able to steer.
- The **arrow object**: parenting the truck, the arrow actor exists for simplification purposes and in this simple simulator embodies the actual main actor. Remote controlled, navigates the game arena.
- **Camera** parented to the tilting chassis: source of visual sensory input to the external program.

3 GAME LOGIC

Each actor needs a set of sensors, controllers and actuators to interact with the game arena. Both input and output from Blender to the external program (the jAER, in this case) and back need be taken care of.

An *always* sensor assigned to the active game camera (parented to the truck) interacts with a python script actuator (see Table 1), responsible for capturing image frames. For convenience, the camera is also assigned the *Escape* key actuator to quit the game.

The arrow object is in control of the car, and receives input both from within Blender -for testing purposes- and from the external program (Table 2). For direct game interaction within Blender, *key* actuators are used, which link to simple *AND* controllers and then each to a different *force* actuator, which simulates acceleration.

Enabling control of the arrow object from the external program was a bit tricky. There are two parts involved: the *property changed*

Table 1. The logic blocks of the camera object.

name	sensor	controller	actuator
camera	Always, 1/4 tics	camera_feeder.py	
quit	key ESC	camera_quit.py	Game, quit

Table 2. The logic blocks of the arrow object, controlling the car.

name	sensor	controller	actuator
uparrow	key	AND	Motion Force +30 Y
downarrow	key	AND	Motion Force -30 Y
rightarrow	key	AND	Motion AngV -0.8 Z
leftarrow	key	AND	Motion AngV +0.8 Z
turn_right	Prop. changed	AND	Motion AngV -0.4 Z
turn_left	Prop. changed	AND	Motion AngV +0.4 Z
throttle	Prop. changed	AND	Motion Force +10 Y

sensors (Table 2) and the modification of such properties via the camera *python script* controller (*camera_feeder.py*, Table 1). In this way, at every game logic time tick, the properties of the arrow object are changed following the commands provided by the external program, and their change fires the controller and thus the desired *force* actuators, resulting in truck motion within the game arena.

4 CAPTURING THE GAME'S ACTIVE CAMERA FRAME IN A PIXELS ARRAY

The camera is parented to the truck and provides the driver's viewpoint. The game screen will be exactly the camera view. From a python script set as a controller, such view can be captured for image processing purposes. In the case of the truck, each pixel must be compared with the same pixel at the previous game logic tick, and a value computed to determine if a pixel changed event must be fired or not.

To capture what the camera sees, we need first the bounding box of the game screen. Since such box will not change during the course of the game, its bounds are stored in a global variable 'Rasterizer.AB_bbox'. Using global variables assigned to python modules is the only way to keep game state throughout game execution, since every time a controller calls a python script, the latter is newly initialized.

```
try:
    b = Rasterizer.AB_bbox
except:
    # allocate 4 integers to capture the box
    #     (x,y,width,height) of the GL_FRONT
    b = Buffer(GL_INT, 4)
    # capture the GL_FRONT bounding box
    glGetIntegerv(GL_VIEWPORT, b)
    Rasterizer.AB_bbox = b
```

To capture the camera frame, we allocate a new buffer with the proper dimensions and then pass it to OpenGL's glReadPixels function. In this case, in black and white (*GL_LUMINANCE* flag).

```
# select the front buffer (the game window)
glReadBuffer(GL_FRONT)
# allocate a buffer for the image
pix = Buffer(GL_BYTE, b[2] * b[3])
# fill the pix array taken from the box
glReadPixels(b[0], b[1], b[2], b[3],
                GL_LUMINANCE, GL_BYTE, pix)
```

For performance purposes, the image contained in the pixel buffer is scaled to a maximum of 128x128 pixels using a C function *scaleImage*, wrapped in the custom python module *EventGenerator* and accessed from within python.

5 GENERATING SYNTHETIC EVENTS

In the simulation, the camera generates a full frame at every time tick, which differs greatly from the asynchronous generation of contrast changed events by each of the silicon retina's arrayed light sensors. The best possible aproximation is to generate a set of events for each frame, all equally time-stamped. The asynchronous nature of the jAER will handle them just the same.

The module *EventGenerator* contains the *createEvents* function, which takes the new pixel buffer and the previous matrix of events, and generates the new matrix of events. Doing so in C is very convenient not only for live game performance, but also for creating and packing the events themselves. Each event requires some computation and byte shifting operations which are way more straightforward in the C language than in python.

Finally, the returned matrix of events is stored in a python module variable, just like the bounding box of the game screen, for pixel comparisons with the next video frame.

6 INTERACTING WITH THE EXTERNAL PROGRAM VIA NETWORK SOCKETS

For ease of use, communication with the external program is handled by a network socket. In this fashion, not only can Blender interact with programs written in languages other than C and python, but also they may run in different computers.

The python server is created by the camera python script controller, and stored into a module variable.

To send information packets through the socket, special attention is required to the format. The *struct* package provides the means to clearly specify the endianness and data types sent.

```
# send events to the client jAER
now = int( (time.time() - start_time) * 1e6 )
# above, in microseconds
packet = ""
# create one compound packet with all events
for e in events:
    # pack into a char string of 2 bytes
    # (a short) and 4 bytes (a long)
    # as standard, no byte padding
    packet += struct.pack('!HL', e, now)
```

7 CONCLUSION

Blender's game engine comes as a great platform for physics simulation, and offers all the power and ease of use of python and C to make the best out of it. As demonstrated, interacting with external,

dedicated software works like a charm once all game logic bricks fall into place.

8 ACKNOWLEDGMENTS

Thanks to Toby Delbruck for his patience in explaining the details of the Monster Truck and the jAER during the Neuromorphic Engineering meeting in Telluride, Colorado, 2007. Deep thanks to Daniel Fasnacht for detailed explanations and heavy debugging of the python network server, and to Yulia Sandamirskaya for working out the driver logic (those differential equations!) in the jAER.

9 APPENDIX: TIPS AND TRICKS

The radius of the parent objects rules above those of any of the children. In our case, the radius of the arrow object, which is parent to the entire truck, must be large enough and positioned just so, for the arrow not to fall by gravity and thus sinking the truck way into the arena when the game starts.

The ground or action arena must be set as an actor, but with no *dynamic* flag: it will be immobile, static. Otherwise it would fall infinitely, and everything else with it. For the expected result, the bounds of the arena must be set to those of its triangle mesh.

Since python scripts are executed newly every time a game logic tic calls the controller, game state can only be stored either in files (undesirable) and as newly created variables on game logic module. To ensure the same variable is used, the script captures the variable in a try and except statement; if the variable is not present, the raised exception is cached and the variable (the server, the bounding box, the event matrix) is initialized.

To avoid potential name conflicts with variables stored on modules, it is advisable to prepend an unusual tag to all variables, creating a *de facto* name space (note the prefix *AB_* to all variables I assign to game engine modules).

Objects from a Blender scene get prepended an *OB* tag to their object names when accessed by python scripts from within the game engine. For example, the arrow object becomes the OBarrow object.

Within the Game engine, objects show the color and shadings of their materials but not any of their textures. For objects to display their texture, the latter must be of type *image* and properly UV mapped to it.

10 RESOURCES

- jAER: Address Event-based Representation protocol implemented in Java. http://jaer.wiki.sourceforge.net
- Source blender file and C/python source code files available at https://jaer.svn.sourceforge.net/svnroot/jaer/trunk/blender/
- Monster Truck at the Neuromorphic Engineering Workshop, Telluride 2007. https://www.neuromorphs.net/ws2007/wiki/monster

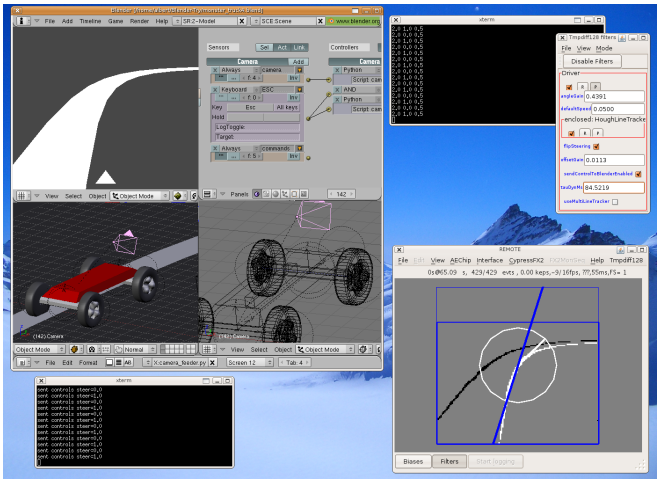

Fig. 2. Blender game engine running on the top left window, and the jAER event viewer and driver control panel on the right.