PySoldier - A 2D Shooter in Python

Computer Game Prototyping

Michael Francker Christensen s031756
Ask Hjorth Larsen s021864
Supervisor: Michael Rose

DTU - Technical University of Denmark

10 November 2005

This space intentionally left blank

Table of contents

Chapter 1

PySoldier - A Realtime 2D Shooter

1.1 What is PySoldier?

PySoldier is a 2D sideways-scrolling shooter. The player controls a soldier by means of mouse and keyboard, and the view is centered on that soldier during normal gameplay. The player can walk around in a world consisting mainly of polygonal platforms, and the objective is to kill as many opposing soldiers as possible, each of which is controlled by another human player across a network.

1.2 Overview of PySoldier

When launching PySoldier, first pyui is initialized and an `Application` is constructed which uses a pyui `Frame` to display a menu. The main game loop is the `run` method of `Application`, which repeatedly invokes the `draw` and `update` methods of pyui along with a custom method which will be specified later, depending on whether the running session of PySoldier is a client or server. Initially this method does nothing.

If the user enters an IP address and selects the *join* option from the menu, PySoldier will attempt to connect to the specified IP address. If connection is successful, the game will be set to run *client mode*. If the user presses the *create* button, the game will run in *server mode*.

In client mode, all user input from the mouse or keyboard which correspond to game controls will be sent by UDP to the server which handles it. The client will constantly update the world with information received from the server.

In server mode, UDP datagrams containing game updates will be sent across the network very frequently, and each such update contains all relevant information in the game. The notion of *relevance* is clarified in section 2.6. For every frame, the client will read any information received from the network

Chapter 2

Architecture and Design

2.1 Overall architecture

PySoldier consists of several separate components managing client behaviour, server behaviour, physical simulation and graphics. The design of different components will be shortly described below, and ultimately in detail in the other sections of this chapter. Furthermore, PySoldier relies on a number of third party libraries, namely Twisted, Hoop, PyUI and PyGame.

2.1.1 Initialization

During startup, PySoldier loads the setup of the physical simulation and creates a map, see section 2.3.Then PySoldier initializes windowed display with and presents a menu to the player. At this stage, the game enters the *main loop*, which runs while the player considers the menu options. The player can now either create a game, enter an IP to a textfield in order to join a game, or quit.

Creating or joining a game will result in the game entering *server* or *client* mode, respectively, and the game will from this point continue running until the player quits. The game loop will behave differently depending on the mode.

2.1.2 Main loop

The main loop will repeatedly perform a number of updates, each to be described now. First the display is updated, see section 2.4. Second the physical simulation will be updated, using a timer to measure the elapsed time between frames. This is described in section 2.2. Next, a custom update is performed which initially does nothing. When a mode is selected, however, this step will update server or client, along with polling for mouse and keyboard input. The effect of mouse and key input differs, depending on whether the game is in client or server mode. In client mode, input will be relayed to the server, while in server mode it will be applied directly to the avatar of the local player - this is described in 2.6.

2.2 Physical simulation

The Hoop library provides most of the functionality needed in PySoldier. This includes collision detection and some actual mechanical simulation. While these things will take care of many details, there are a few flaws or needs in the Hoop library that complicate the design quite formidably.

2.2.1 Collision detection

When designing a collision grid, the basics of which are not to be discussed here(see [1]), there are two different approaches which have each their merits and disadvantages. The simplest approach, which is used in Hoop, is to register each sprite in exactly one square of the collision grid. In case the sprite overlaps other squares, we have to check for actual collision against all the sprites resident in all adjacent squares, totalling 9 squares. Suppose now that there exists a kind of sprite which is larger than a collision square. If two of these sprites meet, they may easily overlap physically, but the collision detector will not detect this if their overlap occurs outside the squares adjacent to those in which they are centered. Thus, no sprite may be larger than the a square of the collision grid.

The other approach avoids this problem by registering a sprite in all those collision squares which overlap with a bounding box of the sprite. Checking for collision in this case requires only consideration of those squares which the sprite overlaps. Needless to say, when sprites overlap several squares in the grid, it can be rather computationally costly to move them, but in most cases, particularly for small sprites, the amount of checking of adjacent squares will be smaller than in the above approach. More importantly, there is no longer any restriction to the size of sprites.

The ulterior motive of this discussion in terms of PySoldier is that the Hoop approach does not support sufficiently large colliding sprites to properly include *terrain* in the simulation. While PySoldier wishes to use large blocks of terrain in the simulation. Since in most cases, only relatively few sprites are moving around at a time, some reverse engineering of the Hoop classes may solve this problem. However such reimplementation is hardly within the scope of this project, and instead all terrain modelling has been done by means of smaller chunks. The complete procedure is described in detail in section 2.3.

2.2.2 Collision handling

Write about all the getting stuck trouble in Hoop

2.2.3 Newtonian movement - sprite dynamics

PySoldier attempts to model newtonian movement of soldiers and bullets. The fact that neither is susceptible to rotation (since the PySoldier perspective is not top-down) simplifies matters quite a lot.

The most important single sprite is the soldier. Affected by gravity and friction, soldiers behave differently when they are in the air or on the ground. Friction in the air is quite small and laminar (i.e. proportional to the velocity of the soldier). Friction on the ground is physically the basis for movement, of course, but the question of how to deal with soldier movement remains biological. Our approach is to simply apply a constant force, let's call it the *motor force* T, which is responsible for propulsion, and a force proportional to the soldier speed's speed v (equivalent to friction again), meaning that a soldier will accelerate with a constant rate at first, then exponentially approach the maximum speed depending on the particular constants chosen in the simulation. In other words, Newton's second law is

$$M\frac{\mathrm{dv}}{\mathrm{dt}} = T - \mu v, \tag{2.1}$$

where M is the soldier's mass and μ the friction constant. Note that the linearity of this equation, which basically governs movement both while the soldier is in the air and on the ground, ensures that the description remains physically correct when extended to two-dimensional movement. It is easily shown that the maximum speed obtained in such a system is exactly

$$v_\infty = \frac{T}{\mu}. \tag{2.2}$$

When a soldier jumps, which is possible only while on the ground, he is simply assigned a specific vertical speed. While in the air, the soldier can still be controlled slightly (this helps climb obstacles), but this extra control must not allow the soldier to obtain superhuman speeds because of the low friction. Thus, the *air control factor* α is introduced, and the motor force is proportional multiplied by this when the soldier is in the air. Selecting

$$\alpha = \epsilon\left(1 - \left|\frac{v}{v_\infty}\right|\right) = \epsilon\left(1 - \left|\frac{v\mu}{T}\right|\right) \tag{2.3}$$

The variable ϵ is chosen to make gameplay good. This choice of air control function will completely eliminate the air control when the speed is near maximum walking speed, while air control is quite high when the speed is low (which is usually the case when a player tries to jump around between obstacles). Of course, the concept of air control has no physical meaning and is only introduced to improve game play. This is done in most games which rely heavily on jumping.

2.2.4 More on sprites

Three other mobile sprites exist, which will now be described briefly:

- Bullet. Spawned at the end of a gunbarrel when the gun is fired. Will continue moving in the gun barrel's direction, but is slowly deflected by gravity and a slight air resistance[1] (future implementations could include wind resistance, which is quite easy to add). Bullets cannot collide with each other, yet on any other collision they will cease to exist. If a Bullet collides with a Soldier, it will deal damage proportional to its kinetic energy $\frac{1}{2}Mv^2$. When the game runs in client mode, however, bullets deal no damage since this is managed through the network.

- Gun. Purely graphical effect, except for the direction in which it points, which determines the firing angle. Each soldier is equipped with a Gun sprite. Generally the gun points in the direction of the mouse cursor.

- Angel. Upon death of a soldier, an angel is spawned. The most prominent feature of the Angel is that unlike most other physical objects, it curiously accelerates upward under the gravitational influence. Angels move through any obstacle and cannot be controlled. After a few moments they disappear, spawning the soldier whose life was previously cut short, thus completing the cycle of life.

Further there is a number of terrain sprites, an approach necessitated by the matters discussed in the previous section. These will be described in section 2.3.

2.3 Map and level building

2.3.1 Introduction

The world of PySoldier consists of soldiers, moving objects, and the level itself. The level is a representation of physical world in which the battle is fought. As such all current level objects consists of stationary objects which can not be penetrated by other physical objects and therefore restricts movement and lines of fire.

2.3.2 Level problems

For making a level there are some issues which need to be adressed before the manufactoring of a level can begin:

- Levels need to have a logical build-up to make the gameplay fair for all sides.

- The level buildup must be compatible with the collision detection.

The game world is a coordinate system with a width of 1200 and a height of 600. The 2 spawn points is placed in each side of the world to ensure a certain time before the players start blasting. In this way most combat will take place around the center of the world. A good level builder will make sure that the spawn points is somewhat isolated from the combat center to ensure that a player can not be killed within the first couple of seconds in a fight.

The collision detection in pygame is using a collisiongrid. Each collisiontile has a defined witdh and height (w = worldwidth / 20, h = worldheight / 20). This makes a total of 400 collision tiles in the world. When an object is added to the world it is registrered in the tile in which it is placed and all adjacent squares. In order to validate collision detection the size of colliding objects in game is restricted by the size of the world.

2.4 Graphics

Most of the graphical details in PySoldier are delegated either to PyUI or Hoop. The Hoop library allows sprites to be equipped with images, and the Hoop engine can render these to the screen. Each sprite in PySoldier uses a category class, which points to an image file. Rotation and translation will be taken care of by Hoop itself, depending on the locations of the sprites in the physical simulation. Graphics in general is not a focal point of the PySoldier development, and only few different sprites are therefore available. More graphical content could be added in later versions, along with e.g. a background image.

[1] Physically, the air resistance should be proportional to the square of the magnitude of the bullet's speed, since the flow of air would be highly turbulent, but this approach generally makes trajectories more boring.

2.5 Peripherals - mouse and key input

2.6 Network architecture

Since PySoldier is a realtime game where players each control one avatar directly, it is quite important for the playability to ensure low latencies. This immediately suggests use of the UDP protocol for the majority of the network traffic. Selection of the protocol is quite important early on, since the entire network code will ultimately depend on this choice. While the UDP protocol is quite fast, however, it is unreliable. Certain one-time events, such as the death of a player, cannot be simply handled by, for example, sending a *player death event*. If the responsible package is lost, the game will be out of synch immediately. Handshaking and distribution of information such as nicknames would further require some kind of guarantee of delivery. This means a TCP/IP protocol might be considered for this kind of events. In PySoldier there is one important detail: since each player only controls a single person, only relatively small amounts of bandwidth will be necessary to even send the complete state of the game. Our initial approach would be to rely *solely* on UDP, and send (possibly redundant) information for every game update. This initial approach allows for two later optimizations:

- Use of rare *large* and frequent *small* network updates, so less bandwidth is used to send information which is unlikely to have changed, or changes so slowly that strict synchronization across the network is unnecessary
- Introduction of TCP/IP to manage one-time events, thus eliminating most redundant data transmission

While these optimizations are valid, we do not plan to implement either unless bandwidth becomes a problem (presently no such problems have been observed during LAN or internet play).

2.6.1 Client/Server model

PySoldier uses a client/server model, where one player creates a *server session* and runs the game locally. This session runs the complete physical simulation, and the observations of the server session are final (i.e. a person dies if the server thinks this is the case, even though clients may not have seen this).

A number of clients may connect to the server, spawning *client sessions* which listen for and handle input from the server. In return clients will send data such as keyboard and mouse input to the server, which the server will parse and apply to the simulation. The server returns game states, i.e. player positions and other data. The exact nature of this data transmission will be discussed in section 2.6.4.

2.6.2 Approaches to client-side interpolation

As mentioned earlier, because of the unreliability of the UDP protocol, we cannot hope to ensure that an exact simulation takes place on the clients. The server will issue game updates with a certain frequency, but in practice, sprites on the clients will only be *close* to their server-side positions.

Apart from sending game data with a high frequency, it is also customary to help the clients preserve a reasonable representation of the server game state by means of guesses or interpolation. For example, if server sends only the positions of sprites (which completely determines the game state), the game play might be seen to stutter on the clients. If the server sends the sprite velocities as well, clients may linearize and simulate player movement which will not only make the game play look more smooth, it will actually - on average - constitute a better approximation to the server representation.

In other words, the advantages of partial client-side physical simulation are two-fold: the simulation will appear more smooth on the client, and will stay closer to the server's representation. PySoldier clients will therefore run a full-featured simulation of sprite movements, only neglecting to apply damage and so on, which will be managed through the network. On reception of an update from the server, a client will in the present version immediately overwrite its physical data with the newly available. It might be reasonable to smooth out this correction by adjusting the client's physical data over a few frames, but this feature remains as yet unimplemented since gameplay progresses reasonably smoothly. Some games will also, in order to further improve the perceived responsiveness on clients, make the client move slightly as soon as controls are pressed,

then silently apply corrections afterwards. This is mostly useful in 1st person perspective games when the player is, so to speak, closer to his avatar, and we have chosen not to implement this.

2.6.3 Client limitation

The present state of the PySoldier does not allow more than one client to connect. This decision was made because only a quite limited amount of computers with appropriate Python software installations were available during development, and the two-player approach eased testing considerably. Care has been taken to ensure that this does not significantly impede the later implementation of multiple players. Basically, the server spawns two players on start-up, the local player immediately taking control of the first one and starting the simulation. The connecting client will take control of the second, but the game may run indefinitely without any clients connecting. Further development of the game would do well to stay with this approach for as long time as possible due to the value of single-player testing.

2.6.4 Game protocol

This section will finally state exactly which information will be sent between client and server. The client will send only input from the peripherals, i.e. mouse and keyboard. Each update consists of a series of 1's and 0's, each indicating whether a certain button is presently pressed. These buttons are: arrow keys up, down, left and right (for movement), and the left mouse button (firing). Finally, instead of sending the two coordinates of the mouse cursor, from which the server would be able to infer the angle in which the client's avatar's gun should point, this angle itself is sent.

The packages sent by the server to the client consists of both soldiers in the game (easily generalized to n players). Sending a soldier means sending position, velocity and health. Also this update would include the frag counts of each player, and the client may infer from the changing of these values that someone has been killed (this may seem inconvenient, but it solves the problem of one-time updates over UDP connections quite splendidly). The server will also send whether or not it is firing, making it customary for the client to simulate the actual bullets.

2.6.5 Consistency

As stated in the previous section, neither client nor server sends the actual positions of bullets. This is partly because the task of keeping track of every bullet created and destroyed would inflate bandwidth requirements significantly. Bullets also move quite quickly, and such information would not travel well across the network. In the selected approach the server will, obviously, simply apply the client's control input to its own model and thus create bullets where appropriate in the simulation. The client has to do exactly the same thing, but it is not in charge of the simulation. The client therefore simply creates the bullets where it thinks they should be, but does not allow these bullets to deal damage to players. The question remains of how consistent the client's representation is. Since bullets move quickly, it can be difficult for the human client player to see whether bullets actually hit, and small inconsistencies will therefore be virtually invisible. For large latencies, however, the game play will be seriously degraded.

One alternative approach would be to send information about the creation of bullets only once. This would leave open the possibility for bullets not appearing due to packet loss, but since the server will consistently manage the damaging and killing of players, this would not seriously affect game play. The possibility this remains of reverting to this approach, should the currently selected approach be unsatisfactory.

2.6.6 Conclusion

The PySoldier network model relies on a UDP client/server structure where the server runs the final physical simulation, while clients run a similar simulation yet apply corrections received from the server with high frequency. Presently only one client is allowed. The client sends almost only mouse and key input to the server, which applies this to the simulation.

Chapter 3

Implementation

3.1 Network implementation

PySoldier uses the *Twisted* network framework for both client and server implementations. Basically, the `Server` and `Client` classes extend the `DatagramProtocol` class of the Twisted framework. Both classes have an `update` method which is polled from the main loop of PySoldier, and which will check the Twisted `reactor` for network input, then (if enough time has elapsed since last time) send an update data to the counterpart.

Two ports are used for communication in PySoldier: 8004 and 8005. All traffic from client to server uses the former, while the latter is used for all data going the opposite way. Presently, all data is sent to a specific IP (i.e. not broadcast) since the test computers could not always be made to connect while using broadcast. In other words, this approach is more likely to function well with firewalls.

The `Client` and `Server` classes have constructors take as parameters an object representing the game world, and the two port numbers used for reading and writing. Data will be read or written to and from the world object when updates are received or sent. Furthermore, the `Client` constructor takes an IP address, which it will connect to.

The behaviour of both these classes is determined by only few methods, which will be described in turn.

3.1.1 The `Client` class

The `update` method, which is invoked from the PySoldier main loop, will check all pending datagrams using `reactor.runUntilCurrent` and `reactor.doSelect(0)`, then check with the `pyui` timer if it is time to write an update to the server. If it is (presently, if more than 0.05 seconds have elapsed since last time), the `writeUpdate` method is invoked.

When updating through the `reactor` object, the `datagramReceived` method is invoked for each datagram arrived since last time. This method first checks whether the IP is equal to the server IP, then forwards to the `parseDatagram` method, which will create an `Unpacker` object from `xdrlib`. The `Unpacker` is used to unpack two `Soldier` objects by means of the `unpackSoldier` method, then the number of frags of each player is unpacked and applied to the game model. If these frag counts increase, the appropriate soldier will be killed by setting the simulation object's `alive` flag to 0.

The `unpackSoldier` method simply reads the position x and y components, the velocity components, aiming direction and health of the soldier, then loads those values into the world simulation. The position components are, importantly, sent as floating point numbers, since otherwise precision loss may result in the soldier colliding with other objects of the simulation. The other values are not as important, and are packed as integers.

Last, the `writeUpdate` method simply polls the `pygame` mouse and keyboard states, then checks which of the PySoldier control keys are down. Each of these will be packed, by means of an `xdrlib` `Packer`, and sent, using the `transport` object on the `Packer`'s buffer.

3.1.2 The `Server` class

The `update` method of this class performs quite similarly to that of class `Client`. First update using the input from `reactor`, then invoke the `writeUpdate` method.

writeUpdate will like before create a Packer . It will pack the information of two Soldier objects by means of the packSoldier method which is analogous in type and order of packing operations to the previously discussed unpackSoldier , only it packs numbers instead of unpacking them. Last, the frag counts of each player is packed, and the buffer of the Packer object is sent through the transport object to all connected clients (the number of which is, as stated previously, limited to one).

3.2 Level implementation

We have chosen to build the present level around a building block structure. The designer has 5 different kinds of block to put anywhere in the world. A dirtblock is a SimObject which is immobile, and with a hit method returning 0 to ensure they can overlap.

The current game version only have one level.

When building the world a list is loaded for each type of dirtblock with (x,y,dirtType). The totalTerrain list is then created adding all terrainlists together and the method populate(x,y,dirtType) generates the whole level. Each block have a width and hight value stated in the blocks source. For GL to make this work with the *.png file we need to have a symmetric image around the center, this is ensured with the code in the source-files as follows:

```
from OpenGL import GL

name = "quad"
image = "texture1.png"
centerX = 0
centerY = 0
numFrames = 1
w = 10
h = 10
points = ( (-w,h), (w,h), (w,-h), (-w,-h) )
primitives = [ (GL.GL_QUADS, (0,1,2,3)) ]
```

The width and height of the object then totalts in $(2w, 2h)$. Each dirtType has its own source file containing the characteristics of the specified object. For illustration in game the different dirtTypes have unique color codes so the user can see the current level buildup. The dirtblocks can overlapse each other to make more distinct levels, like building lego. The process of building a level is hard this way since each block needs to put in place by the designer manually. Given more time an easier way to store level data should be implemented.

In order to avoid collision detection errors the size of the level objects has a restriction as mentioned in "level intro". The collision tiles have the (w,h) = (60,30). Each object is only registrered for collision in the 1 tile and all adjacent squares - This means that the combined width or height of 2 colliding objects can't be greater than the width or height of a tile times 2. This restricts our max width of 2 objects to 120 and the height to 60. Since we dont have to worry about non-moving object in this equation we could make dirtobjects as large as (120 - soldier.width, 60 - soldier.height). Given a soldiers charateristics we get (120 - 14, 60 - 20) = (106, 40), therefore our largest dirtobject has width of 60 and a height of 40 which borders are maximum.

Chapter 4

Testing

Chapter 5

Conclusion

5.1 References

References

[1] Game programming with Python (Game Development Series), Sean Riley