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 written in the Python programming language. 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 rectangular platforms, and the objective is to kill the opposing soldier, which is controlled by another human player across a network.

This document describes in detail the development of PySoldier into a reasonable computer game *prototype*. In other words, the game is not intended to be *complete* as such, yet our development has of course attempted to provide reasonable playability and eliminate any serious bugs, such that the feasibility of the game's ideas can be proven.

The immediately following section will define the rules and goals of PySoldier precisely. The general game architecture is treated in chapter 2, while chapter 3 will go into further detail with some of the more important or difficult implementation issues. The experiences gathered during our own playtesting are discussed in chapter 4, which will also state some of the future features planned or wished for. This chapter will also list all presently known bugs. Last, chapter 5 will conclude on the project as a whole.

A screenshot of PySoldier can be seen on figure 1.1.

1.2 Rules

PySoldier is a 1on1 basic shooter in 2D mode. The player will see the world from the classic platform game perspective. For the game to be a serious prototype we have several modules that must be implemented in order to consider the task complete. First of all we need a physical world in which objects can move around under a newtonian system, this means implementation of gravity, mass for moving objects and forces working in both game dimensions. The rules of the game is quite simple from a player aspekt of the game.

Network is essential since it can only be played using UDP You need a mouse and keyboard for playing the game Python needs to be installed with following packages in the most recent versions:

- hoop

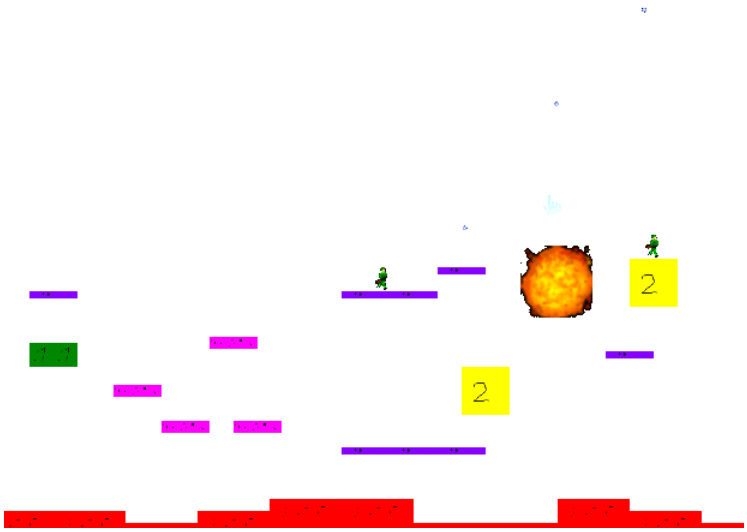

Figure 1.1: Screen shot of PySoldier. One player is firing into the air while a grenade explodes. Note that the background colour has been changed to white from black to ease printing.

- PyOpenGL
- pygame
- PyUI
- twisted
- GLUT

When playing the game the player should move his/her horisontally avatar around in the world using the arrow keys on the keyboard. The game will not allow normal illegal actions as moving up a vertical wall. For movement in vertical dimension the up arrow is the jump button which allow the player to jump a certain height - thus degrading the control in the horisontal line. For actual combat the mouse buttons will be considered the primary input. The game will support a simple point and click interface with a shot being fire in the direction of the mouse cursor from the avatars position. All shots fired will either be removed from the game hitting game obstacles or hitting a player thus causing damage. On death the avatar will momentarily be removed from the game until respawning, and a *frag* is awarded to the opposing player. A simple indicator will display the frag count for both players. A more formal list of rules:

- Two players each control one soldier by means of the controls below
- Soldiers may move in horizontal line by left, right arrow
- Jumping by up arrow
- Left mouse button will fire shot in the direction of the mouse pointer

- Right mouse button will throw a grenade in the direction of the mouse pointer
- Grenades explode after some time
- Soldiers will take damage from hits by shots or nearby explosions
- Soldiers may pass through other soldiers
- Soldiers may not pass through terrain objects
- Bullets and grenades will pass through other bullets or grenades
- Bullets will be removed from the game upon collision
- Grenades bounce off solid surfaces
- Soldiers, bullets and explosions adhere to Newtonian physics
- A frag is scored upon death of the opposing soldier
- At any time, the player with the highest score is considered winning

1.3 Primary evaluation

For the game to work as intended several problems needs to be solved. The manner of the implementation is, as previously mentioned, a *game prototype*. This means that most of the single modules of the game may not be particularly well optimized, but should function *qualitatively* as if in the final implementation. It is easily predicted that the most important overall aspects and hardest to implement will be, in order of implementation time:

1. Creation of a functional physical representation.
2. UDP networking - client/server modes with synchronization.
3. Designing levels and additional game content

For the physical world implementation, the *SimObjects* of the hoop library will be used and each update method will make sure physical laws are upheld. The use of SimObjects will be extended to the level design as a level will be made from impenetrable, stationary objects. Networking will be based on the twisted datagram protocol, and the goal for the network code is to achieve smooth gameplay and responsiveness on clients by means of a sufficiently high frequency of network updates. Finally, most rendering managed again by hoop, which works on top of PyUI and pygame.

1.4 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 2.5. For every frame, the client will read any information received from the network

Chapter 2

Architecture and Design

2.1 Overall architecture

As mentioned in the previous chapter, PySoldier consists of several separate components providing different functionality, particularly network client behaviour, server behaviour, physical simulation and graphics. The design of these different components will be shortly summarized below for a quick overview, and ultimately in greater detail in the then remaining sections of this chapter. Figure 2.1 contains a schematic of the complete game structure.

2.1.1 Initialization

During startup, PySoldier loads the setup of the physical simulation and creates a map, as described in section 2.3.Then PySoldier initializes a windowed display and presents a menu to the player, the details of which are described in section 3.4. 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.

If the user wants to join a game created on his own computer, which is highly useful for debugging when only one computer is available, he should join the 127.0.0.1 IP address, which is for the same reason the default value. Simultaneously hosting and joining a game is equivalent to running two sessions on different computers.

2.1.2 Player management

Upon creation of the world it becomes necessary to track the players in the game and control the interaction between players and avatars. The total state of the game is encapsulated within an *environment*, which thus contains references to physical game world and manages the states of players, in particular their scores. Immediately two *players* are initialized, and each receives a soldier along with a *controller*. The controller is responsible for passing control input to the soldiers, whether this input comes from keyboard and mouse input or network. The controller will be examined closer in section 2.4.2.

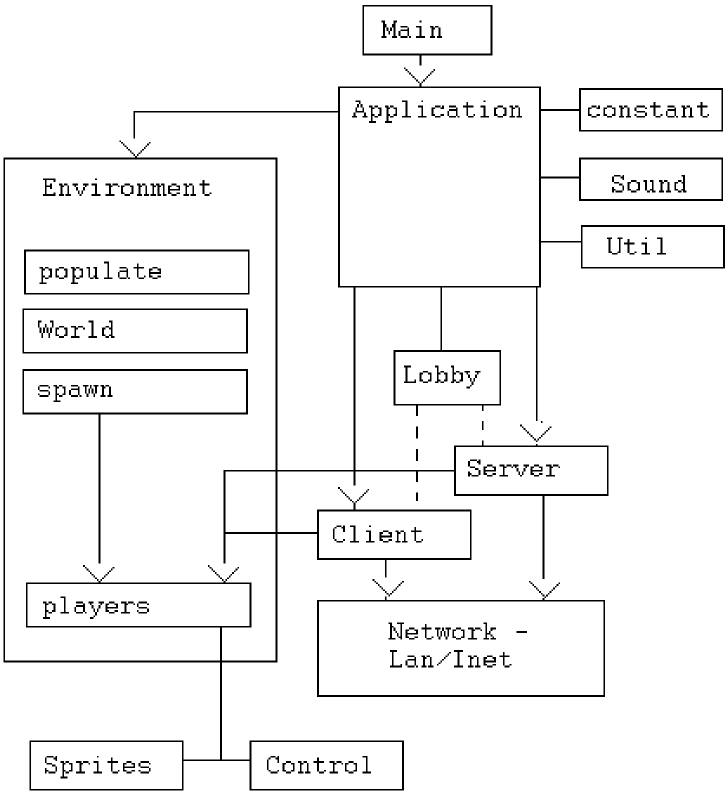

Figure 2.1: This schematic shows the overall structure of PySoldier. All modules are listed except certain trivial classes e.g. containing sprite data. The PySoldier structure attempts to delegate as much functionality as possible to different modules in order to keep different parts of the game mechanics separate.

2.1.3 Main loop

The main loop will repeatedly perform a number of updates, each to be described now. First the display is updated, see section 2.4.1. 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 2.5.

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 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

The hoop library has a default collision handling behaviour which can easily be overridden to support more complex interactions. In PySoldier, when a soldier hits an obstacle, the sprite should not bounce off, but instead stay in contact with that surface. For example, the sprite should rest on a horizontal surface while sliding off vertical ones. While the implementation of sloped surfaces would certainly be beneficial for game play, the hoop library cannot deal with these easily, and we have therefore decided to use only axially aligned terrain. The precise details of this problem are elaborated in section 3.1.

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 actual 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

$$M\frac{\mathrm{d}v}{\mathrm{d}t} = T - \mu v, \tag{2.1}$$

where M is the soldier's mass and μ the friction constant. Note that the linearity of this equation, which in different forms 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. These conditions are all satisfied by this formula:

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

The variable ϵ is chosen to make gameplay good (it is 0.4 presently). 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

Five 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.

- Grenade. These objects are spawned similarly to bullets and obey the same physics although a lot heavier. They generally have lower velocities and do not cease to exist upon collision. Instead they bounce off surfaces realistically, until they reach their preset time limit at which point they spawn an Explosion object at their location and disappear.

- Explosion. This object exists for less than one second, and while it exists, any soldiers inside its collision radius will be propelled away violently while receiving high damage. Remember: grenades don't kill people, explosions kill people.

[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.

- 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.

2.2.5 Firing

When a soldier fires, bullet objects should be created at the location of the soldier's gun's muzzle. If this location resides within the bounds of the soldier sprite, that soldier will immediately hit himself with tragic consequences, unless something is done to prevent it. A reasonable way is to simply define the muzzle point as the closest possible point such that soldier and bullet sprites cannot overlap. Bullets in PySoldier are therefore made to spawn around the soldier on a circle, the radius of which is slightly greater than the sum of the radii of the soldier and bullet objects.

This opens the possibility of bullets spawning on the other side of thin objects, if the muzzle radius is too large. In the current release, the radius fits rather tightly around the soldier sprite, and this problem is not observed in practice for bullets. Grenades, however, are large enough exhibit this problem under some circumstances.

Once the bullet is spawned, it is possible that the soldier is updated before the bullet, moving into his own projectile and taking damage. A simple way to fix this bugs would be to register the spawned bullet with the firing soldier as an argument and make him invulnerable to his own shots. This has not been implemented since we believe that a player shooting wildly into the air - then getting hit by his own bullets, should be severely punished! The present version of PySoldier does not exhibit this problem unless the game stutters (perhaps due to external processes straining the computer), in which case the time interval between updates can become arbitrarily large, so soldier move steps can have any size. This means the problem fundamentally cannot be removed in this way - it is only possible to minimize it by changing the muzzle radius.

2.3 Map and level building

2.3.1 Introduction

The world of PySoldier consists of moving objects, most importantly soldiers, and the terrain 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.

We do not consider extensive level designs an important requirement, but we do implement a default level in order to complete the gameplay. Proper level designs are obviously an important area of any future development.

2.3.2 Level problems

When 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 registered in the tile in which its center is located. In order to validate collision detection, the size of colliding objects in the game must be restricted by the size of the world.

2.4 Interface

2.4.1 In-game 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. Remember that a screenshot can be seen on figure 1.1 in the introduction to this document.

2.4.2 Peripherals - mouse and key input

During each game update, we can poll mouse and keyboard for input and apply it to the game state. If the game is running in client mode, however, it should not be directly applied to the game state, yet instead forwarded across the network to the server.

Also, soldiers may receive instructions on what to do in two different ways, namely from the peripherals directly or through the network. In order to transparently accomodate these differences, we have decided to split the peripheral updates in separate steps: first, the input is read using the PyUI and pygame frameworks, then handled in different updating methods depending on whether the game is in client or server state. In client mode, this consists merely of sending the data across the network as described in section 2.5. In server mode, the method which polls for input will apply the input data to the local soldier's *controller*, an object associated with each soldier which keeps track of what keys are currently *considered* to be pressed by that soldier. The trick is that the data of the controller can be manipulated in any way, either through actual key presses (as in the server case) or by reading input from the network. Thus, for the programmer writing the behaviour of soldiers when reacting to control input may poll the controller object and blindly obey its data, regardless of whether this data originates from the network or the local player, and regardless of whether the game is in server or client mode. Effectively, the input handling has been split into two steps.

2.5 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).

2.5.1 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 2.5.4.

2.5.2 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 of PySoldier *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 without. Also, this particular approach can be dangerous - for example if the client misses a few packages and the sprite hits a corner which blocks its movement, the sprite might get stuck while it should actually proceed around the corner. 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 when response is returned from the server. 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.

2.5.3 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.

2.5.4 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, angle of aim 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.

2.5.5 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.

2.5.6 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

The last chapter dealt with the overall design of the PySoldier components. This chapter will go into detail with the major implementation issues of each component of PySoldier. The immediately following section deals with the physical simulation, which, as shall be seen, presented some serious unexpected problems. Afterwards the map implementation is discussed, followed by the networking modules, which are described in particular detail. Last, the relatively small areas of user interface and inclusion of sound effects in PySoldier will be briefly treated.

3.1 Implementation of physical simulation

All sprites in PySoldier extend the hoop-class `SimObject`. The hoop library manages most of the simple sprite movement and collision detection. The particular parts of the PySoldier simulation that have to be implemented manually are therefore movement behaviour and collision *handling*. These things are generally implemented by overriding the `update` and `hit` methods of `SimObject`. Actually, `Soldier` is derived from a subclass `ImprSimObject` of `SimObject` which had its `update` method slightly changed to support a richer collision handling behaviour.

3.1.1 Movement

The Newtonian movement of `Soldier` objects is implemented by adding three attributes, `xForce`, `yForce` and `mass`. For every frame, during the `update` method of `Soldier`, the two former values are calculated by adding contributions from physical interactions and user input. When no more forces work on the `Soldier`, the total force is divided by the `mass`, multiplied by the time interval supplied as argument to the `update` method by hoop, then added to the components of the sprite velocity, `velocityX` and `velocityY`. The `Update` method of the superclass is then invoked to finish the updating operations.

One last technicality in the movement code is the necessity to distinguish between when a soldier is on the ground and in the air. Simply letting a collision with the ground occur each frame does not work well with hoop, because sprites will generally remain stationary if they collide, and redirection of movement on collision is difficult as the next section will show. The easiest test is simply to check whether placing the soldier a certain distance below the current position would result in a collision with a stationary object. Thus, standing on the ground equates to being sufficiently close to it. One problem may rise with this implementation: if the player hits the ground and is stopped by the collision detector, but this distance is larger than the threshold defining whether the soldier is on the earth, then the soldier will not hit the earth. However, if the collision resets the soldier's speed to 0 (which is the correct behaviour as specified in the game rules) on the collision, gravitation make the soldier slowly approach the surface, eventually descending below the threshold in a matter of very short time.

One minor hindrance with the hoop `hit` method is that it does not distinguish between *probing* a location for obstacles to test whether a unit fits in a certain location, or actually trying to move the object to that location - any collision check will result in `hit` methods being called if other objects overlap with the desired placement. However, by returning 0 from the `hit` method, one can suppress collisions. When performing checks for whether a soldier is in the ground, therefore, are forced to use a somewhat unappealing approach of setting a flag, then checking that flag in the `hit` method to see if we are actually colliding or just testing.

Bullets and grenades move under the effect of gravity and air resistance. The implementation is identical to that of the `Soldier`.

3.1.2 Collision handling

Next, the `hit` method is implemented. The particular desired behaviour is to determine the direction of the surface with which the soldier collides, then slide along that direction without bouncing off or standing still. A simple implementation would be as follows:

1. Try to move toward the desired location.
2. If a collision is detected, try moving in the x direction only.
3. If this too fails, try moving only in the y direction.
4. If all this fails, do nothing.

However, hoop allows only one such try (technically, one invocation of `checkCollide` from the `update` method), and if this fails the object will have its hit method called, but won't move in this frame. The obvious behaviour of letting a soldier on the ground collide with the ground once per frame is thus not a possibility, because the soldier would simply get stuck. Recursively retrying move commands as suggested above from the `hit` method is unappealing, since it would require keeping track of what options had already been tried. To overcome this problem we have decided to create the aforementioned subclass of `SimObject`, called `ImprSimObject`, which simply overrides the `update` method with an almost identical one, trying in turn the three different directions if movement fails.

3.1.3 Implementation problems of sloped curves

It should be noted that the approach which tries moving along each axis is rather crude, seeing as it supports curved surfaces quite poorly. Presently the `findHitDirections` method inherited from `SimObject` is used to elegantly treat collisions with axially aligned objects, but this method would have to be rewritten if generalizing to arbitrary surfaces. In conclusion, implementing arbitrary curve alignment into the collision detection is a noble cause, yet highly time consuming since a lot of hoop code would have to be generalized.

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, is equipped with a hit method returning 0 to ensure they can overlap mutually, and is generally immutable.

The current game version includes only one level design.

When building the world a list of tuples is loaded for each type of dirtblock with each element of the form `(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 has a width and height value stated in the blocks source. For simplicity, the center of each image is set to coincide with center locations of the corresponding sprites. This is ensured with the code in the source-files as follows (this is an excerpt from `SourceTerrain1.py`):

```
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)) ]
```

While it may seem intuitive to change the shape of objects by using irregular polygons in the `points` list, this is not supported by hoop and will break the consistency of collision profiles with graphical appearance.

The width and height of the object then totals 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 have the possibility of overlapping each other to make more complex shapes, much unlike normal bricks. The process of building a level is hard this way since each block needs to be put in place by the designer manually. Given more time an easier and more compact way to store level data should be implemented, for example random content generation, automatic generation from a table or from raw pixel data.

In order to avoid collision detection errors the size of the level objects has a restriction as mentioned in section 2.3. The collision tiles have the (w,h) = (60,30). Each object is only registrered for collision in the one tile containing its center - and collision detection traverses only 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 maximum width of 2 objects to 120 and the height to 60. Since we dont have to worry about non-moving objects in this equation, we can make dirtobjects as large as (120 - soldier.width, 60 - soldier.height), since soldiers are the largest moving objects. For a better collision detector allowing larger objects, the hoop classes may be extended or modified, but these considerations are not within the scope of this project. Given a soldier's charateristics we get the largest allowable size of colliding objects (120 - 16, 60 - 20) = (104, 40), therefore our largest dirtobject have a width of 60 and a height of 40 which borders the maximum given our world size (note that we only take this approach because of the hoop limitations; ordinarily it would be inappropriate to rely on these details).

3.3 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 which receive the following 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.

3.3.1 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.

3.3.2 The `Server` class

The `update` method of this class performs quite similarly to that of class `Client`. First it updates the world state using input from `reactor`, then it invokes the `writeUpdate` method if sufficiently long time has elapsed since last invocation.

`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

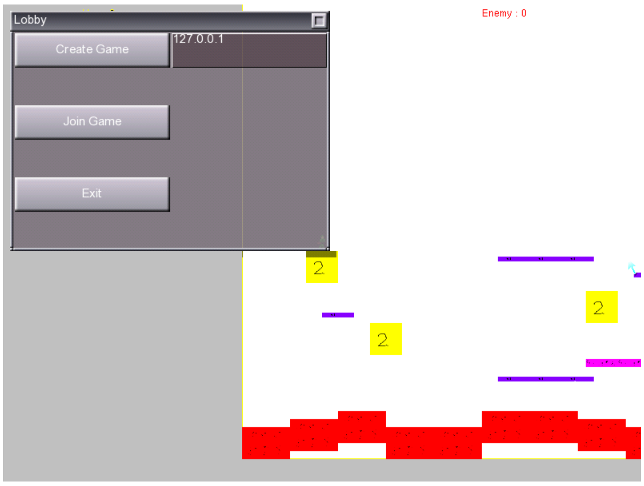

Figure 3.1: The PySoldier menu. The option selected by the user in this menu determines the mode in which the game will run.

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).

3.4 User interface

When `PySoldier` is started the application launches a `LobbyFrame`. The `LobbyFrame` is implemented as a normal frame from `pyui.widgets`, and three buttons and plus a textfield are created and added to it. Each button is assigned a method to call upon activation. The three buttons are named "Create game", "Join game" and "Exit". When using the "Create game" button, the `onCreate` method is invoked, the game enters *server mode* as previously explained, and the player takes control of one of the soldiers, who has already been spawned. Thus, a game will immediately start with the creator controlling one soldier without an opponent present in the game. The application will now start the server and listen for a client to join the running game.

Figure 3.1 shows the PySoldier menu.

In order to join a server which is already running, a client needs to know the IP address of the server he wishes to connect to. Entering an IP address in the textfield and pressing the "Join game" button will invoke the `onJoin` method, making the game connect to a game on the given IP address. If a game is not found, simulation will run in client mode, but the client will not receive any updates from the non-existent server. If a server is at some later time spawned at that IP address, the game will commence appropriately (this indefinite waiting scheme is still used only because it eases debugging). This is because creating and joining servers is less of a hassle without having to retry repeatedly if connection fails initially. Until then the player will not be able to move. If and when, however, a game server exists on the IP address, the client will join the game while taking control of the soldier not controlled by the server.

3.5 Additional game content: sound

In PySoldier 4 different sounds have been implemented. The implementation of these was not planned - more an impulsive move. The sounds are:

- luger.wav - the sound heard when firing
- meinLeben.wav - the sound of a dying man
- theme.wav - the intro music played only once
- explosion.wav - the sound heard when a grenade explodes

The sound files are located in the PySoldier/Sound library. The files are loaded upon starting the game in constants.py using the pygame mixer:

```
fileName = pygame.mixer.Sound('sound/fileName.wav')
```

Each of the sounds is triggered upon an event in the game. The theme song of Monty Python will be played upon creation of the lobbyframe to greet players, while the others are in-game sounds. The luger file is played for every fire command in the game, the meinLeben for every death in the game and the explosion for every grenade. This makes it possible for players to fire shots outside their viewing range and still know if they manage to kill the opponent. The Sound object created by the mixer can be played throughout the program using the code:

```
constants.fileName.play()
```

This makes the wave file play once for each method call.

Chapter 4

Testing and gameplay experience

4.1 Testing

Throughout the project development we have used extensive functional testing. Most of this have been done using ”`print 'whatever you want'`” when testing network or collisions. The use of functional tests paid off mostly in testing collision handling on server and client. For a time we had problem with the clients avatar getting stuck on level objects. This proved to be caused by rounding errors, because we only sent soldier positions across the network as integers - not floats. The integers were then rounded down, and once received on the client, accidently causing the avatar to collide with objects. Simply using `print (soldier.posX, soldier.posY)` we found the avatar to be inside a dirtBlock. Once overlapping the dirtBlock, the avatar's hit method prohibited all moving - thus getting stuck. Most types of debugging have been done on problems like this example.

The game code does not rely on excessive number crunching or complicated loops, and most of the functionality has direct impact on the graphical representation of the game state. The network and collision detection code which would ordinarily require the highest level of testing, is mostly located within third party libraries, and therefore structural testing is not considered necessary. We assume that the collision detection and other modules have been tested thoroughly by their developers. The network is implemented with no degree of freedom whatsoever. Every package sent has a preset length and size. The unpacker upon receiving unpacks the package to the same preset length and size; should an error occur the game would shut down immediatly and the appropriate error message and stack trace would be printed by twisted. Therefore we believe that the performed functional tests still ensure the stability of our project within reasonable margins.

Here is a list of the known bugs:

- Grenades fail to explode if the collision profile of the explosion will exceed the boundaries of the world.
- Grenades can under rare circumstances get stuck in the corners of terrain objects. Some slight corrections in the `hit` method should fix this.
- Grenades can spawn so far in front of soldiers that they may pass through thin membranes which should otherwise block the way. A properly designed level would not allow objects near the edge of the world, thus avoiding this otherwise unfixable issue.

4.2 Gameplay experience

In its present state, PySoldier can be played indefinitely by two players. The basic dynamics of the game work quite well in general with only a few easily fixed known bugs. The network also operates without any serious issues. On the client side simulation, however, game play can suffer especially if the connection is bad. This is and inevitability and no different from other games in the genre. No actual bugs have yet been recorded here either. Thus, physical simulation and networking both seem quite complete. The game, however, lacks some diversity in terms of different levels, more weapons and so on which ensures playability in the long run.

4.3 Further development

Having implemented the game with as many assets as possible in the given time a few of our favourites have been left out in the current version of PySoldier. The most pressing matter would be a working updated physical simulation allowing objects of non- rectangular shape. Rounded curves and triangular shapes would be preferred in the actual game as it would make the game more natural. Sloped curves that could be climbed by the player without jumping can help remove some of the *platform* game atmosphere, making the game feel more modern and smooth. In the game's current state, game play can feel slow due to the possibly excessive jumping around on rectangular boxes.

Other features yet to be implemented include new classes for the players to select. This feature is quite easily implemented, seeing as graphics and numerical data (such as walking speed or jump height) can easily be changed, adding only the complexity of sending the soldier type information across the network as well. Implementation of different kinds of bullets and gun types would be similarly quite easy.

Last, allowing any number of clients to connect at the same time would require some expansion of the networking classes, but it is a reasonably simple task. Note that we prefer at this stage, not to allow this because it complicates play testing.

Chapter 5

Conclusion

In this chapter we will write about the final program, our expectations, further plans for PySoldier and our project development.

The final state of the project is in our minds both lacking and satisfying. The game as first seen in our minds has been realized but still not to its full potential. The game play was intended from the beginning to be fast paced and dynamical, and this has certainly been achieved satisfactorily. We are somewhat disappointed that we could not find time to implement non-rectangular terrain objects, but these difficulties arise from limitations in hoop. This feature along with a deeper physical implementation to allow soldiers to move up and down hills was the next item on the todo-list. Adding different types of soldiers and weaponry, along with enabling a free-for-all mode for more players would be most desirable. Unfortunately much of our time has been spent on studying hoop and twisted libraries to support the current state of the game. The third-party libraries aid the development speed but also somewhat restricts your freedom, at least in the case of the hoop physical simulation.

In these last stages of the development the use of python has been somewhat more clearsighted. Producing games using pygame and other libraries makes development faster as our experience and understanding of python has increased. The one thing which we still find lacking in python is the amount of available documentation. This was particularly problematic for PyUI and certain parts of twisted. Overall we can conclude that the final product is satisfying for a first time experience in python and given more dedication and time it would become a great game. Already at this stage it can be fulfilling enough to jump around shooting.

References

[1] Game programming with Python (Game Development Series), Sean Riley, CHARLES RIVER MEDIA, INC. 2004. ISBN 1-58450-258-4