Tower Defense

Game implemented as a real-time system

Diogo Silva
dbtds@ua.pt
60337

Eduardo Sousa
eduardosousa@ua.pt
68633

Prof. Paulo Pedreiras
Real-time Systems

2016/12/19

Game implemented as a real-time system

Tower Defense

How does the game works?

Tower Defense

The objective of the game is to **not allow the monsters to reach the end** of the path.

The **user** is able to control where to **place towers and sell them**.

User does **not have any control** over the tower or monsters **behaviour**.

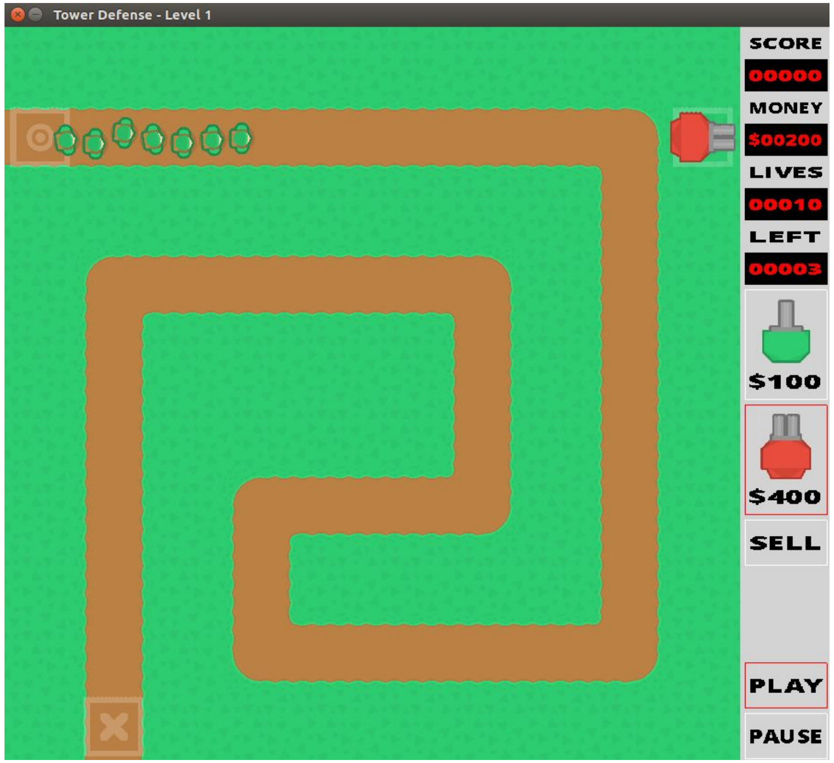

Figure 1: Final result

Game implemented as a real-time system

Tower Defense

How the dynamics of the game work?

Monster sensors and actuators

Sensors (there is noise)

Each monster has **3 eyes**, a left eye at 90 degrees, a middle eye at 0 degrees and a right eye at -90 degrees.

Actuators

They are also able to choose when to **move** forward or **rotate**.

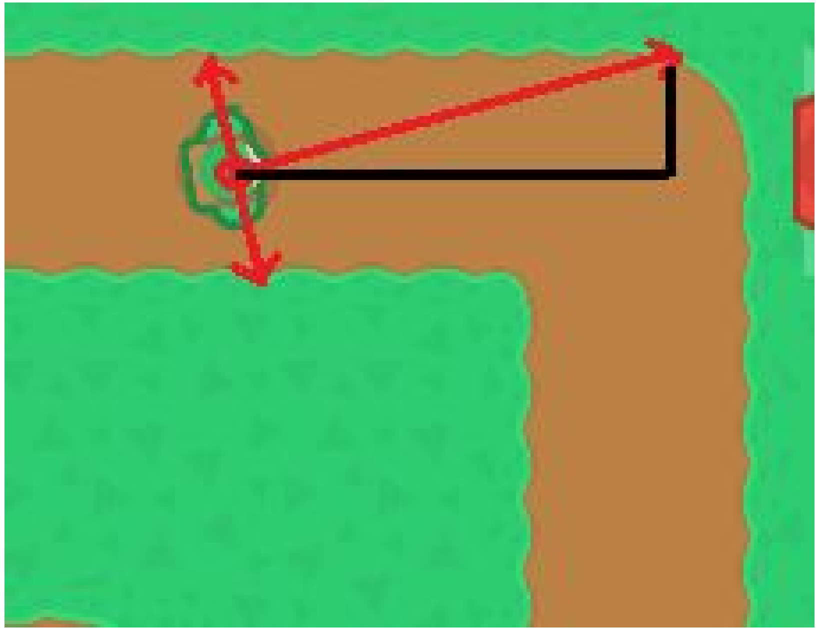
Figure 2: Representation of the robot's sensors

Monster sensors and actuators

Sensors (there is noise)

Radar - Allows find any monsters within a specific range (black circle). Although the bullets will not reach that range (red circle)

Actuators

Direction - Allows the tower to rotate

Shoot - Allows the tower to shoot

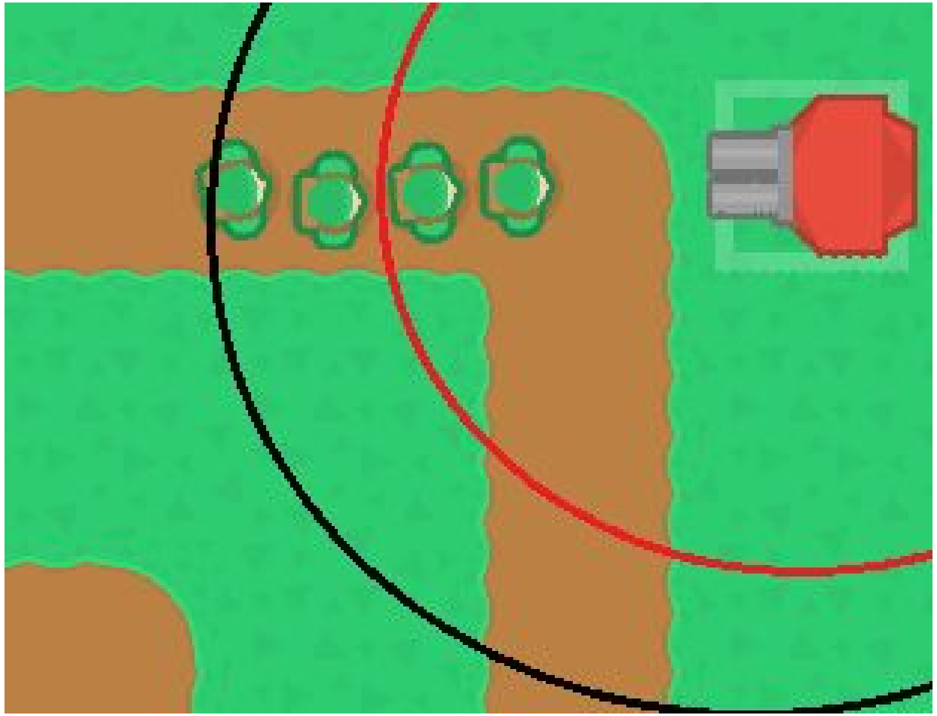

Figure 3: Representation of the tower's sensors

Game implemented as a real-time system

Tower Defense

How are tasks organized?

Tasks Overview

There are four distinct tasks implemented in the system:

- ➔ God which process requests;
- ➔ Tower, Monster and User Interaction that send requests;

None of the tasks that send requests are not supposed to access the World data, they access using an interface.

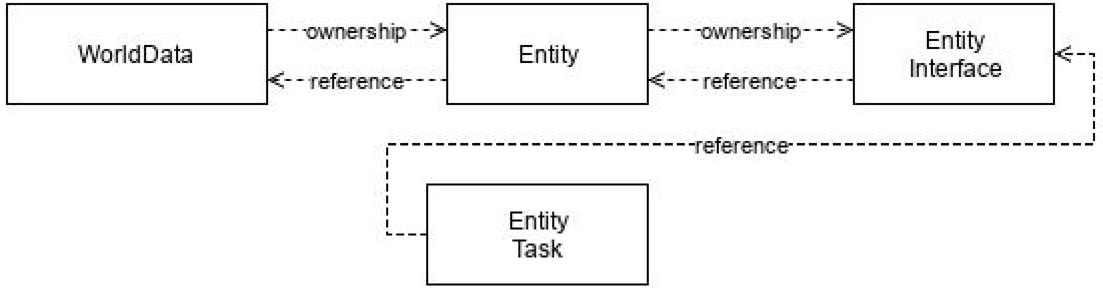

Figure 4: Generalized solution for the world data access problem

Monster task

The monster task is **periodic** and has the following life cycle:

1. **Sense** - Checking information that eyes can see (busy waiting)
2. **Process** the information perceived by the eyes
3. Issue a **request** to move and rotate

Priority: 70
Periodicity: 50ms

Tower task

The tower task is **periodic** and has the following life cycle:

1. **Sense** - Checking information that radar can see (busy waiting)
2. **Process** the information perceived by the radar
3. Issue a **request** to rotate the cannon and shoot

Priority: 70
Periodicity: 50ms

User interaction task

The user interaction task is **sporadic**. That task is responsible for **receiving user interaction** from the viewer and **issue a request for the god task** to apply it.

Priority: 80

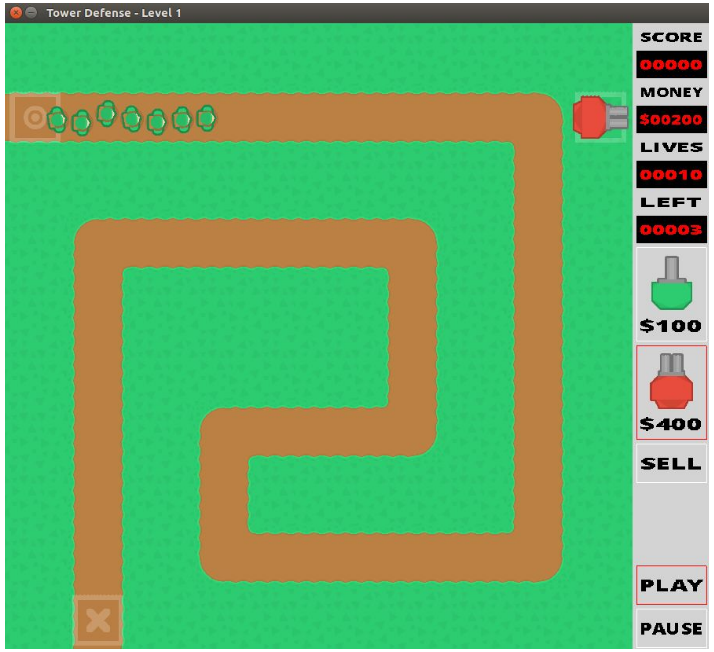

Figure 5: Example of the visual interface

God task

The God task is **periodic** and has the following responsibilities:

- ➔ Process all the **requests** from the other tasks;
  - ◆ Tower and monster actuators;
  - ◆ Interaction on the interface.
- ➔ Ensuring that the laws of the simulated world are not violated;
- ➔ Creates/removes tasks when needed;
- ➔ Send the world state information to the viewer.

Priority: 90

Periodicity: 25ms (1 / 25ms = **40 Hz**, which is important for the visual interface, to have 40 frames per second)

Game implemented as a real-time system

Tower Defense

How does the god tasks control the requests?

Controlling the requests from tasks

Monsters and Towers will **store requests** that receive from their **interface**, that is **controlled by the task**. The same happens with the **User Interaction** but it store requests related with **user clicks**.

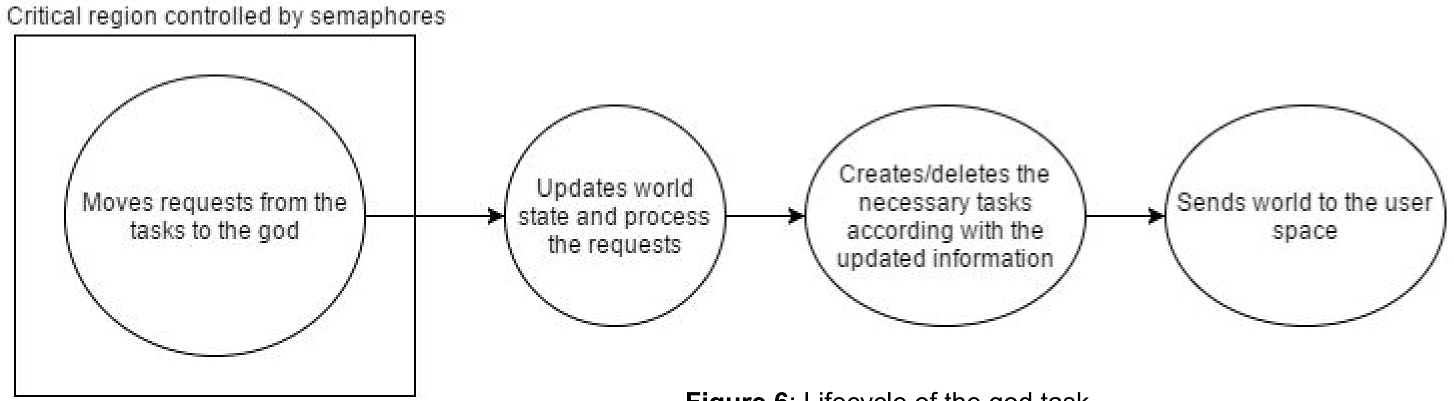
Critical region controlled by semaphores
Moves requests from the tasks to the god
Updates world state and process the requests
Creates/deletes the necessary tasks according with the updated information
Sends world to the user space

Figure 6: Lifecycle of the god task

Game implemented as a real-time system

Tower Defense

How is the data shared?

Data serialization

Data exchange **between the game engine (kernel space) and the viewer (user space)** is done through **real time pipes**.

The **game engine** has a pipe in which it **sends the game information**, so the viewer can show it. The **viewer** has a pipe in which it **sends the user interaction** back to the game engine.

The data is **serialized using the Cereal** library (http://uscilab.github.io/cereal/), which will take care of the process of marshalling and unmarshalling the objects between the different modules the make the program.

Viewer data buffering

The viewer receives game information from the game engine through a **real time pipe**. Since the frequency of the arrival of game information is higher than the frequency at which the viewer can render new information in the screen, the viewer has **game information buffer where it stores the information**.

The game information buffer **stores the game information of the frame that it is being rendered and the newest game information**.

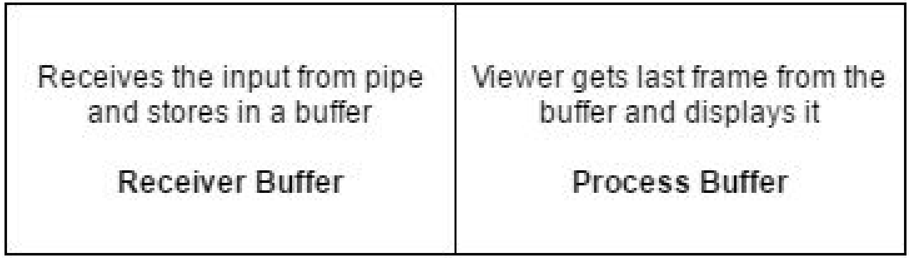
Receives the input from pipe and stores in a buffer
Receiver Buffer
Viewer gets last frame from the buffer and displays it
Process Buffer

Figure 7: Strategy used to receive data from the pipe (double buffering)

QUESTIONS?