AI applied to a simple 2D environment

Michael Gerber, Nicolas Ganz *BMS Zürich*

November 9, 2012

Contents

1 Abstract

Here is some sample text to show the initial in the introductory paragraph of this template article. The color and line height of the initial can be modified in the preamble of this document.

2 Introduction

2.1 Section 1

- First item in a list
- Second item in a list
- Third item in a list

2.2 Section 2

3 Artificial Intelligence

3.1 What is AI?

This question has about as many answers as there are people studying artificial intelligence. Without getting too philosophical one could say that intelligence is defined by the efficiency of patterns used to solve problems in a complex environment and the ability to learn or generate new patterns by observing or testing.

Man-made programs to solve such problems have been around for decades and as algorithm efficiency and computational power increases they get ever more powerful.

An AI usually is directed towards one goal. This could mean solving a mathematical challenge most efficiently, finding patterns in huge amounts of data to categorize new sets of data or even to find smarter ways of learning.

3.2 History of the Field

3.3 Technological Singularity

3.4 Real World Examples

3.4.1 Amazone Recommandations

Everyone knows the “People who bought X also bought Y”-kind of recommandations online shops provide you with. Sometimes they are on the spot and other times they are as far off as they can be. The idea of course is to find patterns in customer interest which can be used to offer people things they are likely to buy.

3.4.2 Google search algorithms

3.4.3 DARPA

3.4.4 Deep Blue

3.4.5 IBM Watson

3.5 Traditional AI vs Neural Networks

3.6 Some AI Algorithms in Detail

3.6.1 Graph Search

3.6.2 Beyas Networks

$$P(A|B) = \frac{P(B|A) * P(A)}{P(B)}$$

3.6.3 Markov Decision Processes

3.6.4 Natural Language Learning

3.6.5 Particle Filters

3.6.6 Heuristics

3.6.7 Game Theory

4 Project

What do we need to create a 2 dimensional jump and run game? First of all we need a physics engine to recreate a realistic environment. Then we need a game engine to build our game on. After that we need the game itself and a environment creator to randomly generate the worlds.

Now we need an AI. But how do we connect the AI with the game? How can the AI controll the player? How can the AI see where the next block is? That's where the Application Programming Interface (API) steps in. With this connection the AI can easily read the environment and move the player. Lastly we "just" need the AI algorithms.

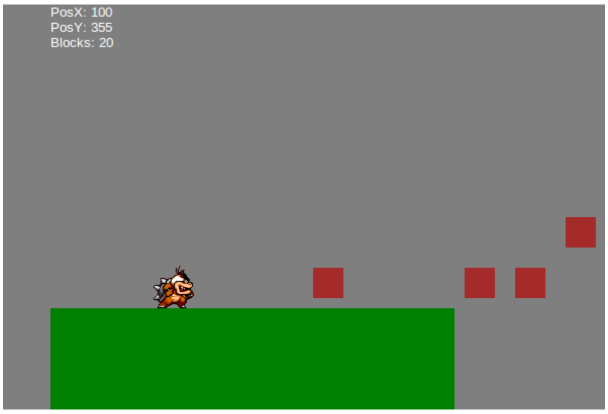

Figure 4.1: *The final game at the start point.*

4.1 Physics Engine

A physics engine is able to recreate an entire physical system like our world. They are mostly common in animated films, scientific experiments and video games. The gravity is the biggest part to cover. There are less and more accurate engines. In some the falling objects get faster or the objects can break when they fall on a solid ground.

4.1.1 Gravity Example

A block gets thrown horizontally with the force F. This block has now the force from the throw (horizontally) and from the gravity (vertically).

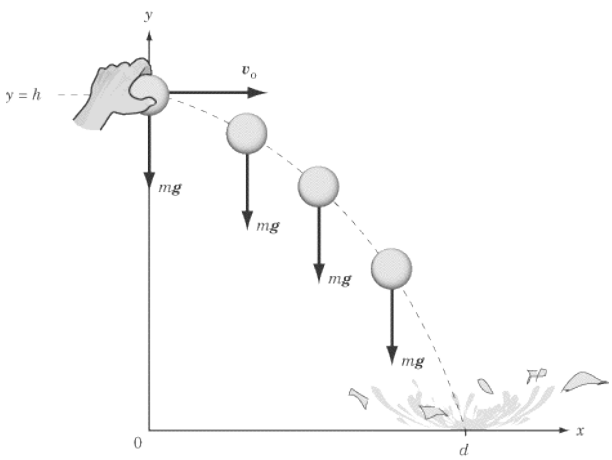

Figure 4.2: *Example for gravity.*

4.1.2 Collision Example

A block gets thrown horizontally with the force F and collides with another, steady one with the same size and weight. After the collision the first block will fly back with half the force, while the other one gets pushed away with the other half of the force.

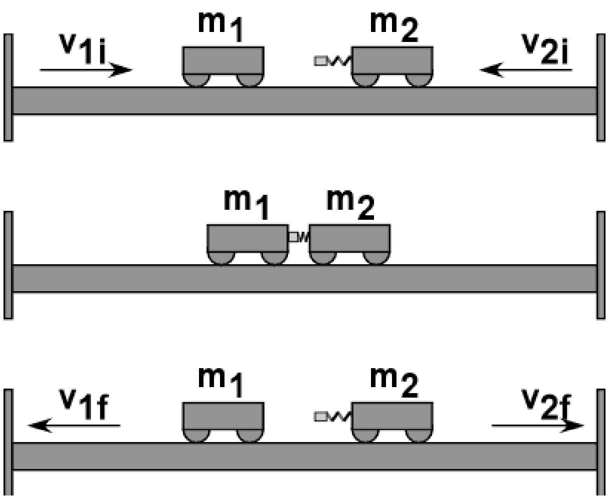

Figure 4.3: *Example for a collision.*

4.1.3 Types of Physics Engines

There are two main types of physics engines. The high-precision and the real-time engines:

high-precision This type of engine is used for exact copies of a physical system. They are mostly common for animations that must be very precise like scientific studies or animated films. It does not matter in this animations how many resources where used while rendering because you can let the machine run for many hours rendering the world and than analyze the result at the end.

To wrap it up: Those engines need much processing powers but are very precise. Quality before quantity.

real-time This type of engine is used for interactive animations like video games. The artificial physics must be calculated immediately. Unlike the high-precision engines this type of engine does not need much processing power, for it is minimized.

To wrap it up: The real-time engines do not need much processing power but are not very accurate. Quantity before quality.

4.1.4 What do we need?

collision We need to be able to check for a collision between the player and a block above the ground.

gravity We need to be able to use gravity to jump and fall down.

real-time We are creating a game, so we need in the first place a fast engine.

4.1.5 Setting Boundaries

But let us stop here. Even though it is a really interesting topic, it is too extensive to go into the details. If you would like to know more, there are plenty of comprehensive studies about physics engines.

4.2 Game Engine

We want the physics engine to create a game. But how can one do this? All by himself? No, we need to find a link between the game and the physics engine. This is where the game engine has a turn. It offers the opportunity never to touch the physics engine and to make the step from a world with just moving objects to a world, controlled by a human.

4.2.1 What does a Game Engine?

- It unburdens the interaction between the human's controls like a keystroke or a mouse click and the animations that can be seen on the display.
- It eases the animations of the objects. For example when the player walks then should the character animate this movement.
- It implements the element of sound. With this feature you can easily add background music and/or sound effects.
- They are mostly specialized on 2D or 3D worlds. 3D worlds are way more complicated but they are more realistic.
- They are either built for a specific platform or they can compile the same source code for multiple platforms. For example if it supports multiple platforms you can create a game and run it on Windows, Linux, Xbox and PlayStation.
- Some engines even include a AI system to help building one. They often helps with the data logistics and sensor access.

4.2.2 Examples of Game Engines

CryENGINE A engine created by Crytek. It runs on PS3, Xbox 360 and PC. It has an advanced AI system with sensors like sight and hearing. The games Crysis, FarCry and Warface are the most famous of many.

Unreal Engine A engine created by Epic Games. It runs on PC, PS3, Xbox 360, Wii U, PSVITA, Android, iOS and Flash. The Unreal AI is also advanced and includes a special navigation mesh system to optimize the performance and memory usage. Dishonored, Gears of War, Batman: Arkham City and Mass Effect are the best examples for this engine.

Anvil Until 2006 it was known as Scimitar and is created by Ubisoft. It runs on PC, PS3, Xbox 360, Wii U and PSVITA. Prince of Persia, Assassin's Creed and Tom Clancy's Rainbow 6: Patriots are the most known examples for this engine.

IW Engine A engine created by Infinity Ward. It runs on PC, PS3, Xbox 360, Wii and Wii U. The complete Call of Duty series expect the first one is based on this engine.

Blender This is the most popular open source game engine.

4.2.3 What do we need?

We don't need a complex game engine for we are creating a simple, 2 dimensional game. It should just cover the basic necessities like:

- interaction between user interface and animations
- animations for the objects
- sound implementations
- built for PC only should be sufficient
- 2 dimensional

4.3 The Game

For it would take too much time to create an own physics and game engine, we decided to use the open source engine Crafty. It is based on the programming language JavaScript allowing the game to run in a modern web browser.

4.3.1 Goal

We want to create a game with this specifications:

- 2 dimensional
- jump and run
- blocks to jump on
- gaps in the floor to fall down
- randomly generated
- generator ensures that the game is solvable
- finish line at the end of the parcour
- die from falling down
- win from reaching the finish line
- after winning or dying: restart the game

4.3.2 Basic Composition

We used a event-driven development architecture. That means that the engine triggers events and we define what should happen.

Here is the main structure of the game in a theoretically programming language:

```
When the player hits a solid object:
  The player stops moving

When the player hits a deadly area:
  The player dies
  Restart the game

When the player hits the finish line:
  The player wins
  Restart the game
```

4.3.3 Animation Sprites

To bring life to our player we decided to use a sprite. A sprite is basically a collection of images with the same character but in different positions. You just have to decide which of them belong to which movement.

Figure 4.4: *The sprite to animate our player.*

standing right If the player stands and looks to the right the game should use the image from row 1 and column 1

standing left If the player stands and looks to the left the game should use the image from row 2 and column 1

jumping right If the player jumps to the right the game should use the image from row 1 and column 3

jumping left If the player jumps to the left the game should use the image from row 2 and column 3

walking right If the player walks to the right the game should switch through all images from row 3

walking left If the player walks to the left the game should switch through all images from row 4

4.3.4 Environment Creator

Our environment consists of blocks and a floor with gaps. We now need a creator that generates random environments. To make it consistent we chose that all blocks have the same size. They are in one of this three heights:

1. The height of the player. Easy to jump on.
2. Twice the height of the player. The maximal height to jump on.
3. Thrice the height of the player. Just reachable from a block from the first or second height.

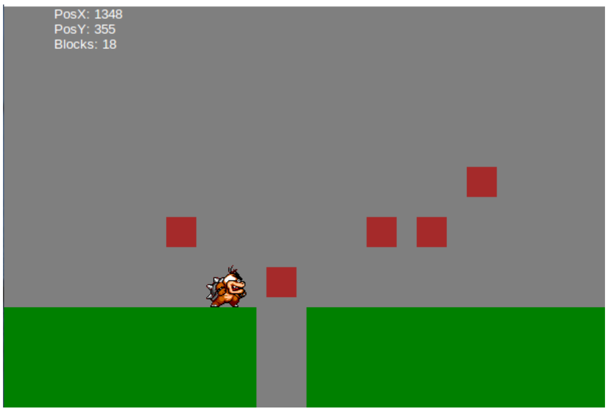

Figure 4.5: *The three possible block heights.*

The gaps in the floor exist in three different sizes:

one block These gaps are very easy to jump over.

two blocks These are a little harder, but should pose no problem.

three blocks These are the most difficult ones. To jump over them you have to stand on the very last corner of the floor or on a block.

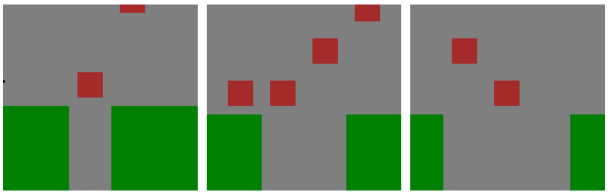

Figure 4.6: *The three possible sizes of the gaps.*

Prevent Unsolvable Environments

Basically we could just say "create some blocks and some gaps in the ground" but it would most likely get unsolvable. To prevent this we have chosen this insurances to prevent that:

- The blocks from the first or second height are no problem for the player to jump on.
- The blocks on the third height may not exist if they don't have an accessible block from the first or second height. This is to make sure that the player can jump on every block.
- The gaps may not be bigger than the player could jump.

Composition

This is the basic composition of the creator in a theoretical programming language:

```
Define the start point
Until the whole width of the field is used:
  Set a random end point
  Create a ground from the start to end point
  Get a random gap width (1, 2 or 3)
  Set the start point: end point + gap width

15 times:
  Until the x-coordinate is unique:
    Generate random x-coordinate

  Get a random height of 1 or 2
  Generate the block from x and the height

  With a probability of 3:
    Generate a next block with the height 3
```

4.4 Application Programming Interface (API)

As mentioned before we need to create an interface for the AI to control the player and read the environment. To gain full control we need actors and sensors.

For the human the actors are the keys and his fingers with which he can handle the player. The sensors are the monitor and his eyes. With this he can analyze what is happening.

Now we have to adapt those actors and sensors to our API.

4.4.1 Actors

The actors are responsible for the actions of the player. We need those available actions:

jump The AI must be able to jump to the left and the right.

walk The AI must be able to walk to the left and the right.

4.4.2 Sensors

The sensors are responsible for the measuring of the environment. We need those available sensors:

won? Did the player win?

died? Did the player die?

my position Where is the position of the player?

nearest block Where is the nearest block?

all blocks Where are all blocks? This is needed to plan a few steps further

nearest gap Where is the nearest gap?

all gaps Where are all gaps?

finish line Where is the finish line? To check if the AI is going in to the right direction.

4.5 Our Artificial Intelligence (AI)

5 Sum up

6 Sources and Glossary

6.1 Sources

Physics Engines

- Physics engine - Wikipedia
  (http://en.wikipedia.org/wiki/Physics_engine)

Game Engines

- Game engine - Wikipedia
  (http://en.wikipedia.org/wiki/Game_engine)
- CryENGINE | Crytek
  (http://www.crytek.com/cryengine)
- Game Engine Technology by Unreal
  (http://www.unrealengine.com)
- Anvil (game engine) - Wikipedia
  (http://en.wikipedia.org/wiki/Anvil_(game_engine))
- IW engine - Wikipedia
  (http://en.wikipedia.org/wiki/IW_engine))

The Game

- Crafty - Javascript Game Engine
  (http://craftyjs.com/)

6.2 Image Sources

List of Figures