Charles University in Prague

Faculty of Mathematics and Physics

BACHELOR THESIS

Štěpán Havránek

3D akční hra v podivném městě

Department of Theoretical Computer Science and Mathematical Logic

Supervisor of the bachelor thesis: Mgr. Tomáš Balyo

Study programme: Computer Science

Specialization: Programming

Prague 2013

I am grateful for all the support I received. I would like to thank my supervisor Mgr. Tomáš Balyo for the help and guidance he has given me. Then I would like to thank Ing. Josef Smutný for introducing me to the basics of programing and algorithms at the high school.

I declare that I carried out this bachelor thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In Prague, May 24, 2013 signature

Název práce: 3D akční hra v podivném městě

Autor: Štěpán Havránek
Katedra / Ústav: Katedra teoretické informatiky a matematické logiky

Vedoucí bakalářské práce: Mgr. Tomáš Balyo, Katedra teoretické informatiky a matematické logiky

Abstrakt: Práce se zabývá návrhem a následnou implementací real-time akční hry ve světě, který se nedá popsat klasickým třídimensionálním reálným lineárním vektorovým prostorem. Tato počítačová hra ovšem používá 3D zobrazování. Práce popisuje celý průběh vývoje: použití různých známých i vlastních technik, algoritmů a datových struktur. Při návrhu implementace různých částí také popisuje rozhodování mezi různýmy způsoby a rozebírá i ty nakonec nepoužité. Součástí díla jsou i umělé bytosti obývající město v tomto prostoru. Dále je zde i protihráč, který se snaží plánovat své kroky tak, aby zabránil hráčovi město obsadit a snaží se to udělat sám.

Klíčová slova: real-time akční hra, 3D hra, umělá inteligence, plánování

Title: 3D action game in a bizzare city

Author: Štěpán Havránek

Department / Institute: Department of Theoretical Computer Science and Mathematical Logic

Supervisor of the bachelor thesis: Mgr. Tomáš Balyo, Department of Theoretical Computer Science and Mathematical Logic

Abstract: This thesis deals with a conception and implementation of a real-time 3D action game. This game is placed in a bizzare surrounding that is not a subset of an ordinary real three-dimensional linear space. The work presents the whole game implementation process. It goes through various sets of techniques, algorithms and data structures used or considered during the development. It also describes different ways of solving specified problems and the choices between them. Moreover there are artificial beings situated in the town inside the surrounding. The player's goal is to capture the whole city. There is also an opponent, who systematically plans his tasks and tries to possess the city as well.

Keywords: real-time action game, 3D game, artificial intelligence, planning

Contents

Introduction

There are many action games with 3D graphical visualisation. The main reason to start using synthetized 3-dimensional space was to bring more realistic feeling from the game to the player. Nowadays developers and designers are trying to make better and better simulations of our planet or real situations in real places using 3D graphics. The aim of this thesis is different from these ideas. We are not trying to display on the screen the same picture you can see when you turn off your computer and go outside. We bring the player a game situated in a space which does not follow basic physical laws of our world. It can be fun but it mainly improves the player's imagination and abstract thinking.

Imagine a game that may look like a classical 3D. It looks like you are in an ordinary town, but parts of the game map are connected to each other as a generic graph. In this game you can go straight until you reach your first position, but you do not come from the back of your original stand at all. For example you can come from the right or from any other direction. And this is the setting of our game.

Player's goal will be to occupy the entire town. He must go to all of the town quarters and capture them one by one. His opponent has exactly the same goal. Because of that both players leave quarters, they have captured, guarded by their friends. The one, who first gets oriented and understands the map and gets all parts of the town to his property, wins

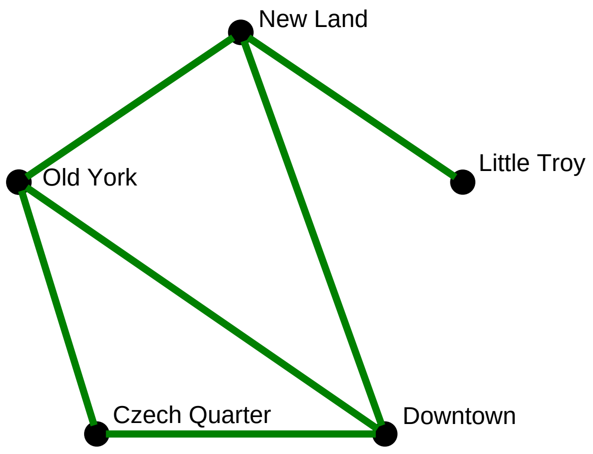
This thesis mainly describes implementation of the whole action game situated in the introduced town. We will begin with description of our software project architecture based on Microsoft XNA Game studio [1]. Then we will go through real-time programming issues, data representation, used algorithms or modified versions of well-known algorithms for our specific case. The last part of programming part will be implementation of AI for player's opponent. During this part we will also point out several areas for further development.

The last part of this work contains user documentation for our product. The reader will find out how to set up and, of course, play the game.

1. The Game

1.1. Detail Description and Rules

The city we are playing in is divided into separate quarters. Each of the quarters has its unique name (ex. Downtown). Some of these quarters are connected to some of the others. Together they form a graph. The town graph is always connected, but there can be a quarter with degree only 1 (see Picture 1).



Picture 1 Town quarters make a graph

Every quarter has somewhere inside a flag or an empty flagpole. The flag indicates who owns the particular quarter. Your flag means that this quarter is in your property. Otherwise the quarter can belong to your opponent or to nobody. Either way you should try to capture the quarter which is not yours. The game begins with one quarter owned by the player and one opponent's. The rest of town is without an owner. The goal of the game is to capture all the quarters in the town. When any of the players has reached this goal, the game is over.

Do not worry about the quarters you have captured. Your guards will gradually appear there. They have only one duty – look after your quarter. When an enemy comes into this quarter, he becomes a target for your guards. The number of guards per one quarter is limited and if you capture opponent's quarter, his guards will stay until you or your guards kill them. We limit the sum of player's and enemy's guards. So if you capture a quarter full of enemy guards, your guard would not appear until you kill at least one of the other.

How to kill somebody? You can always use your hands, but it is not recommended. Killing a person with bare hands is not practical, so we came up with guns. There are lots of gun types you can use. We define four categories of gun availability:

1. For everybody
2. For guards
3. For players
4. Only boxed ones

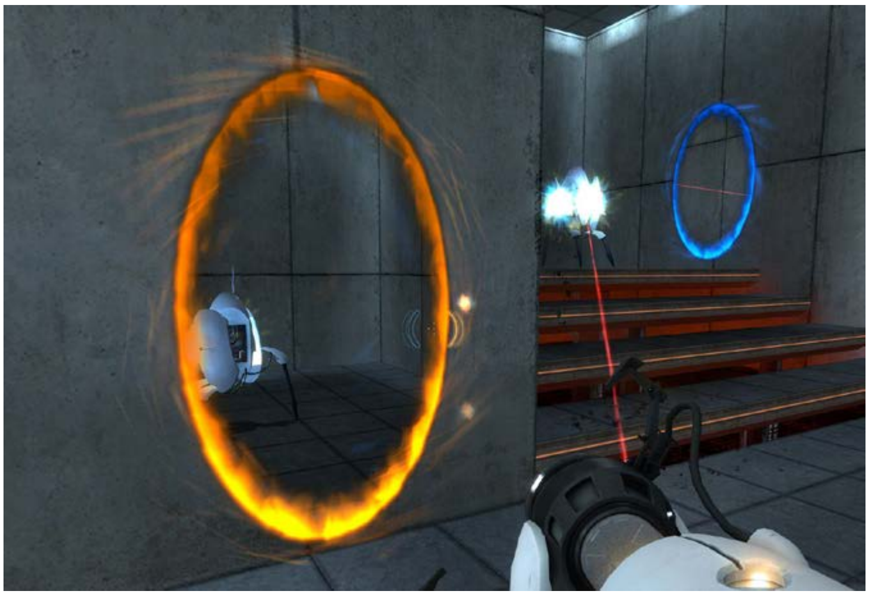
Guns from the first category are held by everyone (including walkers) at the beginning of the game. The ones from the fourth category are available only in boxes which are, if you are lucky, lying on the streets. Note that not only you can take guns from boxes. Your opponent will do it too. The ones from second and third category are simply in the default inventory of the guards and the players.

Since you have guns and your enemies have guns too, it is necessary to use them. You should kill all the enemy guards in the quarter you are going to capture. Then you can safely raise your flag. You will need to kill your opponent several times. Because when one of the players gets killed, he will lose all of his quarters except one, if he has at least one. The killed player will appear alive in the only quarter, he has, over again. If he does not have any quarter, he will show up in some empty one. If you lose your quarters by getting shot, your guards will stay there. Only new ones do not appear anymore.

1.2. Similar Games

There are already games that have something similar with our game. Some of the ideas and rules were inspired by existing games and our game combines their elements together.

Our town, composed from the quarters our way, can be alternatively described as a set of quarters with some kind of portals in the streets. If you enter one of the portals, you will appear in another quarter in front of the corresponding portal. On suchlike portals is based a game named Portal [6]. This game developed by Valve is running on the legendary Half Life engine. It combines action and logic game elements. Portal is a first person shooter (FPS) game and it takes place in ordinary 3D space (characters are inside a factory). What is put as an extra in this game is a “portal-gun”. You have a gun that shoots magic windows on the walls, but you can have simultaneously opened only two these windows (blue and orange in Picture 2). So if you try to shoot a third window on the wall, one of the original windows disappears. These two windows make together a pair of corresponding portals – if you walk into one of them, you will come out from the other one. Moreover you can throw things through the portals. The goal of this game is to use this portal-gun to solve variety of puzzles inside the factory and survive. Our game actually uses more than one pair of corresponding portals.



Picture 2 Portal gameplay

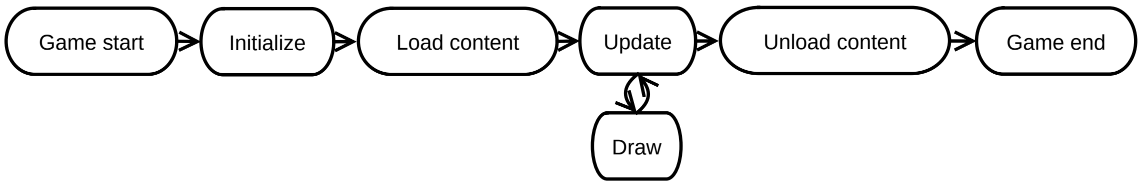
The idea of quarter capturing and guard generating in our game evokes similarity with at least two games that mutually are not similar at all. The older one is a robot war strategy called Z [7]. This game, released in 1996, brings up a map divided into parts represented by a flag and some kind of factory. During the gameplay you lead your units to capture the flags. The related factories, you have already captured, produce new units for you. The second game using similar concept is the continuator of a well-known series – the GTA: San Andreas [8]. Except the large campaign this game offers, there is a possibility to move freely inside the entire map. During one concrete stadium of the campaign, the town (town covers almost the whole map) is splitted into parts that are dominated by various gangs. Much like in our game, the occupied town quarter is guarded by the gangsters. The player is a member of one of the gangs too. Sometimes one gang assaults a quarter owned by another gang. Usually if the player does not come to help to defend the endangered quarter, the quarter will be taken over by the enemy gang. The same attack can be initialised by the player too.

2. Implementation

2.1. *Program Architecture for Real-time Game*

Programming real-time applications is a discipline different to other types of software development. High emphasis is placed on an early response to user input and apparent continuity of episodic process. In other words the game must be able to react and compute its routine at least twenty five times per second. The frequency 25Hz is the frame-rate value that human eye perceives as continuous motion. For example the European standard for television broadcasting (PAL) uses this frequency [9]. Since the process has to be fast we need to do some calculations only approximately or asynchronously. We will use both of these techniques in our game.

Now let's take a look how to make a game architecture for our software. We adopt practices from XNA [1]. XNA libraries provide prepared process model for the whole game (see the model diagram in Picture 3). First we need to initialize our components, then load all needed content. Content loading can be very slow so we should do it before the actual game begins. After loading the main game loop comes. The main game loop is an endless cycle between updating the game logic and drawing the scene. As soon as the game logic decides the game is over, we break the main loop, unload loaded content and do whatever we want. For example we exit the whole application or restart it.



Picture 3 Game life cycle diagram

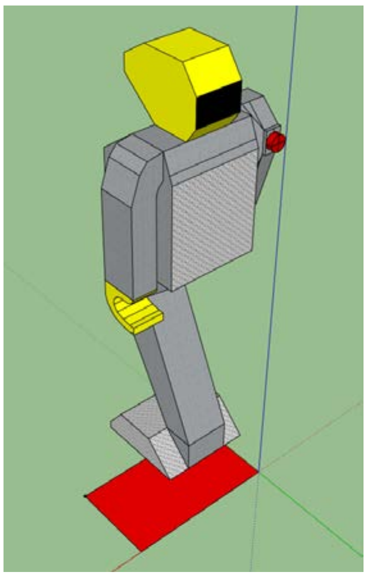
It is good to have this process distributed into separate components. A component model improves the clarity of the whole solution. We have several smaller modules running according to the diagram metioned in Picture 3: Town, Player, Opponent. Moreover the town component distributes these operations into quarters and quarters into walkers, flying bullets, etc. In other words this model allows us to divide computations between components which represent logical parts in our game. Actually this is the idea of object-oriented programming. In the next chapters we describe these objects and components and we will keep the same terminology as we are using in our game source codes which are attached to this text.

2.2. Space and the Game World

2.2.1. Problem Analysis and Basic Geometry

Before we begin modelling the bizzare surrounding as it was defined, we will prepare some basic building elements. We assemble the world hierarchically and up to specific level we can ignore that the result will not be placeable into standard three-dimensional vector space. Moreover on the lowest levels we consider only two dimensions. The third, height, will be added later in and only in selected functionalities, because we do not need it everywhere. Finally the two-dimensional processing will be faster and that is very important to us.

On the lowest level of the space hierarchy we define the following basic geometrical elements: line segment, triangle and convex quadrangle[1]. Everything in our space will be based on the quadrangles. Or more precisely, every object in the game has its projection into two-dimensional space as a quadrangle (visualised in Picture 4). The quadrangle is representing the object's floor projection (ground-plan). These quadrangles are used for collision detection between objects.



Picture 4 Robot and its quadrangle projection (red)

[1] Convex quadrangle (in sources called Quadrangle) is the basic structure in our game architecture. We will often refer it in the implementation chapter. Elements line segment and triangle are only auxilary.

2.2.2. Testing for Collisions

Quite often we need to check if two objects are in collision or not. For example if a bullet hits a man. To get this information we take their quadrangle projection and compute the collision. Our way to do that is to split the quadrangles into two triangles and check them for collisions. This division creates four subprocesses. Now the last thing we need to do to detect the collision is to compute whether two triangles collide. This is simple: we split the triangles into three line segments and find out if any of them is crossing any line segment from the second triangle. This check has to be done for all the three line segments against all the other, so it requires nine subprocesses. We must not forget that, if there is a triangle inside another one, it counts as a collision too. Since we know that borders off these triangles do not collide, at least one of the vertices of the first triangle inside the second triangle indicates that the entire first triangle is inside the second. Deciding if a point is inside a triangle is little bit tricky. We connect the tested point with all triangle vertices by vectors. Then we calculate angles between all the vector pairs. The tested point is inside the triangle if and only if sum of the angles equals to 2π [5]. Moreover if we do not find the first triangle inside the second, we must test whether the second is inside the first.

2.2.3. Game Objects in Bizzare Space

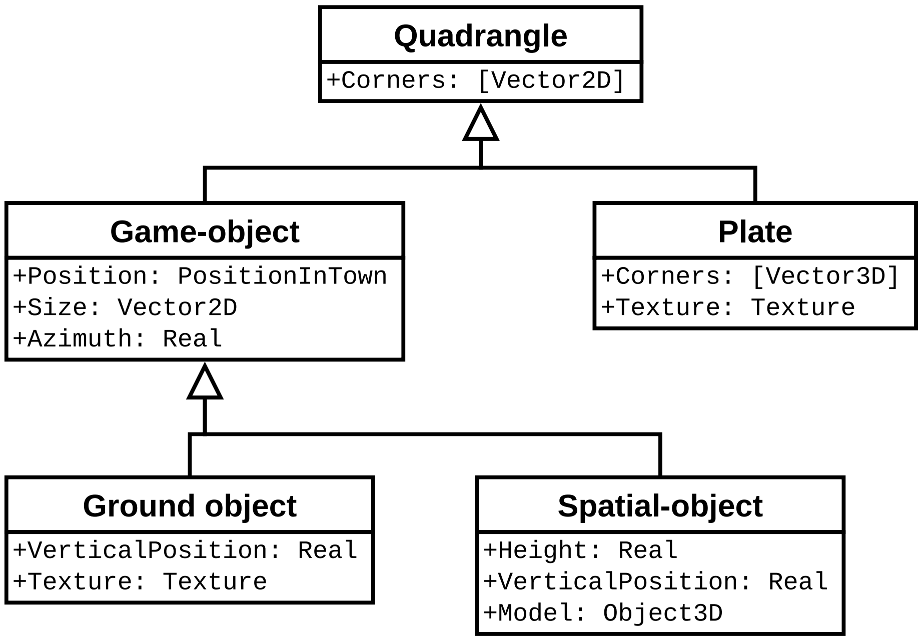
Considering the game logic, the use of quadrangles is not the best way to represent objects in the game. A quadrangle is defined by four points and it can be little bit confusing if we imagine that we have prepared a 3D-object (ex. robot) and we want to insert it into the game. Should we every time somehow adjust the 3D-object to quadrangle we have already defined by for corners? No, using quadrangle in this case would mean that every 3D-object in the game needs to define the position of the base four corners. A better way is to add a next level into space hierarchy. We present the game-object[2]. Game-object is a structure ready to be used with 3D-objects and it is still simple enough to be in two-dimensional space. For work with varied 3D-objects we will use their block shaped bounding box with edges parallel to the axes of three-dimensional space. The bottom base of this cuboid is a rectangle. And this rectangle is represented by the game-object. The game-object

[2] Game-object (in sources GameObject) is the next important component of the game architecture.

carries information about its position, size and azimuth (rotation). The right question here is: what is the position? Is it information about xy-coordinates in simplified two-dimensional space? Or are we now in our bizzare world and position is some kind of description of location in there. The second option is right. The game-object, as the name says, describes the base of every object in the game. So it has to carry full information about location in our result surrounding.

Now it is time to figure out how to represent our bizzare space. After all, what are our technical capabilities? We can display set of objects variously transformed by position in three-dimensional linear space, azimuth, scale and some projection parameters on the screen. So somehow we need to use classical three-dimensional space. The idea is to split our bizzare space into parts which separately are vector spaces. The position of a game-object is specified by the concrete part of the world and a coordinate vector from linear space of the particular part. Now it is clear why we can use only classical linear space on lower levels of abstraction.

The game-object holds basic information about everything in the game. It also provides projection into quadrangles: it takes vector space coordinates from position, size and azimuth and calculates four corners. Now we can implement many of game-object derivations: spatial-objects which are carrying 3D-objects, or flat ground objects and plates for work with only textures instead of 3D-objects (the implemented objects' relations are shown in Picture 5).



Picture 5 Space objects hierarchy

Now we need to decide how to split our bizzare space into separate linear spaces. In the first chapter we learned that our bizzare space actually is a town with the quarters connected to each other as a generic graph. The reason, why is our surrounding so unrealistic, is the fact that the town-graph can be non-planar. The largest part which is still linear space is exactly the town-graph vertex. So we use division by the quarters. Thus a position is defined by a pair of vector and quarter identifer.

From the description above it is clear that we can correctly compute the collision between two game objects only if they are in the same quarter. This should not be a problem at all. All we have to do is to conceive game logic to avoid inter-quarter collisions.

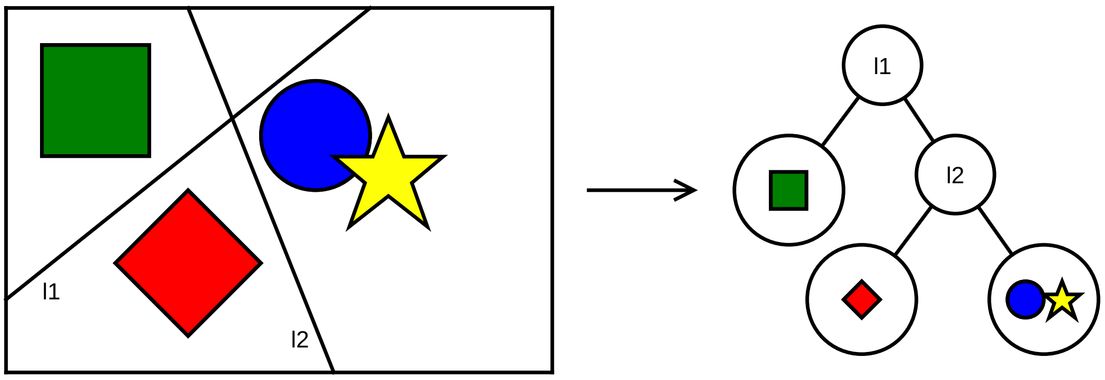
2.2.4. Space Partitioning

There are many objects we need to test for collision against each other in the town quarter. Collision detection has to be run during every update. We cannot afford to miss the fact that two objects have non-empty intersection. Our naive approach, check for collisions all object in the quarter against each other, has quadratic complexity.

$$T(n) \epsilon \theta(n^2)$$

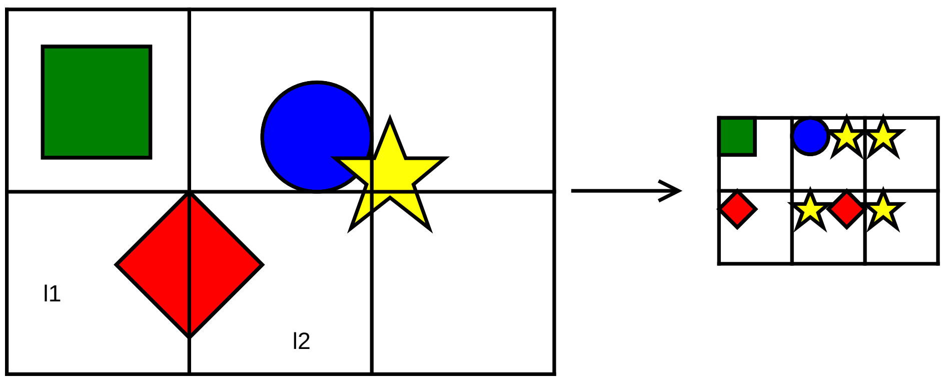
Is this good enough? Actually we do not need to test collision between two objects that are located across the whole quarter from each other. So the idea is to test the collision only between objects that are close together. We need to divide objects into groups by their position inside the quarter.

One well-known technique deals with this problem. It is called space partitioning. There are data structures like BSP Trees or Quadrant Trees used in space partitioning [2]. These techniques are based on search trees. In every node the space is divided into k parts and each part is recursively handled by one child-node. Leaf nodes of the search tree contain objects that are located in the area specified by all the nodes above the leaf in the tree (see Picture 6). The advantage of this data structure is that leaves do not have to be in the same depth. When you need to test collision, you know that only objects from the same leaf can be in collision. Because two objects from different leaves are not in the same part of the space. However moving objects are the weakness of this structure. If the object changes its location and gets out of the area defined by its leaf, it is not easy to find a new leaf that the object belongs to. It takes logarithmical time – when you go through the tree into the deepest leaf, but we are creating real-time game and we need these calculations to be fast. We want to search for the right area for an object in constant time.



Picture 6 Scene represented by BSP Tree

From space partitioning trees we take over the idea of dividing the quarter into areas of objects close to each other, but we will not build any trees at all. We create parts of fixed size formed into a squared grid (see Picture 7). The part the specified object belongs to is simply calculated as the position in the quarter divided by the grid size. Similar partitioning would be the result of Quadrant Trees with evenly distributed objects. Now with our data structure and evenly distributed objects the collision test of all objects in the quarter takes about $k(\frac{n}{k})^2 = \frac{n^2}{k}$ where k is number of grid fields. Do we have evenly distributed objects? We generally use collision detection to avoid two objects to intersect, for example a person cannot go through a building, or a flying bullet kills people. Thus we do not have the objects exactly evenly distributed, but our game logic approximately lays them out so. Moreover this data structure has to avoid us to test collision between two objects far away from each other. If we have a lot of objects close together, we probably want to test them for collisions.



Picture 7 Scene represented by grid partitioning

2.3. Town Generator

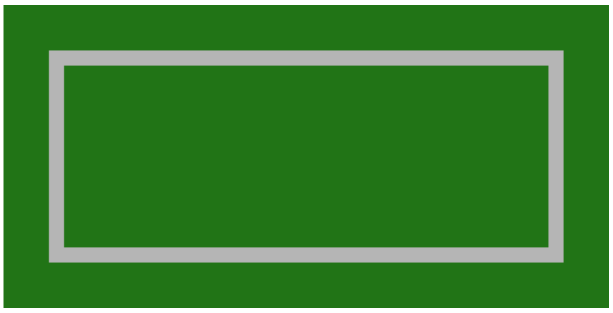
After the content is loaded but before the game starts, we need to create the world where our game will take place. Our game, because of its specific rules, has not any prebuilt maps. For every game instance we will create the whole scene from scratch.

2.3.1. Town-graph

The only input is the number of quarters (n) that will be in our town. We prepare an empty non-oriented graph with n vertices. Into this graph we add edges. Because we want to have this graph connected (one component), we insert a path which contains all vertices first. Without loss of generality we can join by edge always the two vertices which are next to each other in our data representation. It does not matter how we have the vertices ordered. We just need them joined. Also it is not needed to have them in a cycle, so we don't do a cycle – only simple path of length n. The resulting graph could be a regular output, but the game with only this type of map would be boring. We want to add some extra edges into our graph. We just go through all potential edges and use a pseudo-random number generator to decide whether to add the edge to the graph or not. Now we have a graph describing our town. The vertices represent quarters and the edges are joining streets between them.

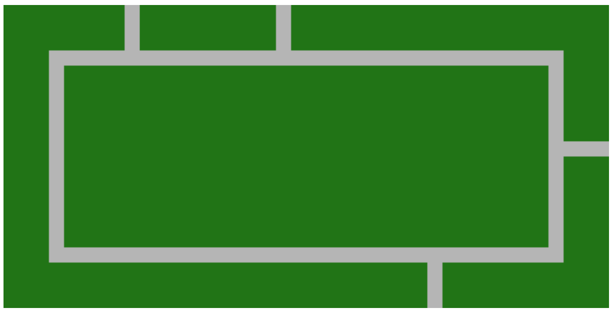
2.3.2. Quarter Generator

Now it is time for creating every single quarter. The only input for quarter generating procedure is its degree – the number of neighbouring quarters. The quarter is placed into a rectangle. First we decide where the interfaces (connections to nearby quarters) will be. We are choosing from top, right, bottom or left side of the rectangle. Then we prepare the road and sidewalk network. Every segment of the road is lined with sidewalk. We start with the "border" road of the quarter (visualised in Picture 8) – smaller rectangle around the quarter formed by road and sidewalk.



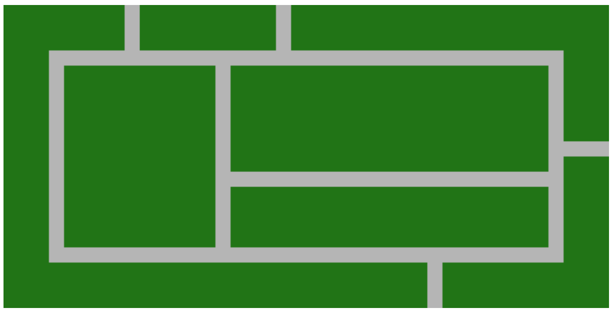
Picture 8 Town quarter with border streets

To the border road we connect interface roads like in Picture 9.



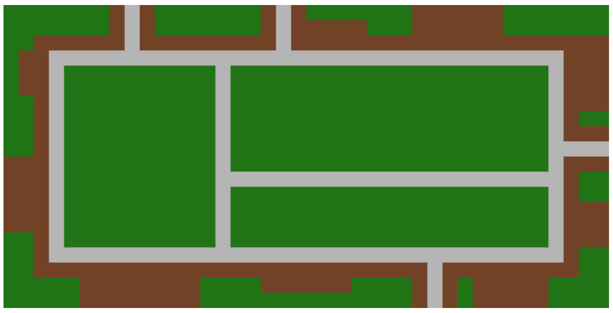
Picture 9 Town quarter with interface roads (quarter with degree 4)

Inside the border road rectangle we have an empty space. Using a pseudo-random generator we will with some probability cross the rectangle by a road and split it into two empty rectangles. This we can recursively iterate. Now we have the road network done (example of road network in Picture 10).



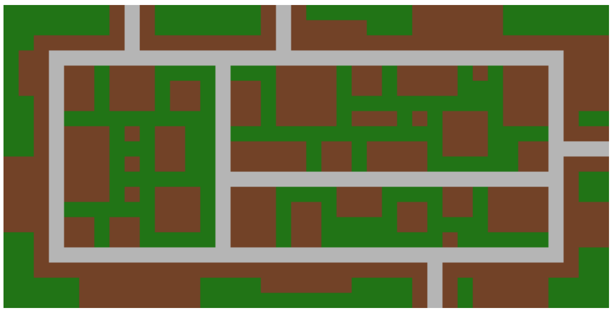
Picture 10 Town quarter with inner road network

Since we have roads inside the quarter we can start putting in buildings and decoration objects. At first we build the border buildings, fences and walls. It is necessary to prevent the player from getting out of the quarter. So we put these types of barriers around the border road and the interface roads with no spaces between the barriers. The result would look like the map in Picture 11.



Picture 11 Town quarter with border buildings, fences and walls

Then we have empty rectangles inside the road network. These we fill with buildings with spaces between them (see filled quarter in Picture 12). Or we do not. We use the random number generator to decide whether to fill the empty space by buildings.



Picture 12 Complete town quarter with inner buildings

2.3.3. Path Finding

We have done all the quarters and its interfaces are joined – every interface has a pointer to its opposite interface situated in the nearby quarter. We analyse now sidewalks and roads inside the quarters. We build a graph that will show paths through the town to future humans. Vertices of this graph are located on the sidewalk and edges are between those which are reachable without collision with building. We specially add interface vertices too. These we connect to the vertices from the opposite interface. This whole town path-graph must be connected so we have to go somewhere through the road, but never through any building.

The main role of the town path-graph structure will be path finding. Whenever the structure gets two vertices, it must find the shortest path between them. That is why we implement the classical A* algorithm [3]. We need a good heuristic for this graph search algorithm. There are two categories of heuristics for A* algorithm. Using V set of vertices, E set of edges, $h: V \rightarrow \mathbb{R}$ heuristic,

$d\colon V \times V \to \mathbb{R}$ real shortest path length, $p\colon E \to \mathbb{R}$ edge price (length) the heuristics are:

- Admissible: $\forall v \in V\colon h_{target}(v) \leq d(v, target)$
- Monotone: $\forall v, w \in V, (v, w) \in E\colon h_{target}(v) \leq p(v, w) + h_{target}(w)$

Because our graph generally has cycles, we need the heuristic to be not only admissible but it has to be monotone too. A* commonly uses distance in Euclid metrics. However our space does not even theoretically satisfy the triangle inequality axiom. We can use Euclid metrics if the target and the ranked vertex are in the same quarter. If they are not, without non-trivial computations we don't know how far it is. The best lower estimate is the distance between two interface path-graph vertices – those from opposite quarters. This distance is constant ($l_{interface}$). We have chosen it in the town generation process (see Picture 13).

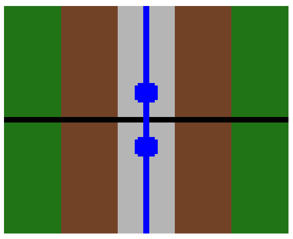
Definition (our heuristic)**:**

$h_{target}(v) = euklidDistance(v, target) \Leftrightarrow v, target$ are in the same quarter

$h_{target}(v) = \ l_{interface} \Leftrightarrow v, target$ are in different quarters

Observation: The above defined heuristic is admissible and monotonous.

Proof: The Euclid distance is admissible because there is no shorter path than the straight line. The price of an edge is actually the Euclid distance between two points in the quarter. So thanks to the triangle inequality axiom the monotone definition is satisfied. When the target is in another quarter, the path just has to go through an quarter interface. So the $l_{interface}$ constant is a valid lower estimate. If we select the neighbour vertex w from the same quarter, the edge price cannot be negative $p(v, w) \geq 0$ and the heuristic is the same $h_{target}(w) = l_{interface}$. If we select w from another quarter, the (v, w) edge is right the interface one (visible in Picture 13), so $p(v, w) = l_{interface}$. And because the heuristic in vertex w cannot be negative, the monotone definition is satisfied in this case too. ■



Picture 13 Two opposite quarter interfaces with the path-graph vertices connected by edge of constant length

2.4. *Boxes, Tools and Action-objects*

In the previous chapter we have built the world: quarters, streets and buildings. Before we create virtual beings or something like that, we need to prepare some kind of interactive content for them.

We will start with tools. Tools will be a generic entity in our game that a human can handle. First we need boxes. Boxes react to collision other way than other non-active content does. Bullet will destroy the box and human will unpack it and take whatever will be there. We make two types of boxes for our game: toolbox and healbox. Healbox is simple, there is some kind of medicine and the human which takes it, will be healed. Toolbox contains a variation of tools.

Tool is generally held by a human and it can do some kind of action. The only way we use the tool abstraction is for guns, but we leave the tool concept prepared for future additions. Each tool has a pointer to its holder so it can take position, azimuth or something else. It also has a "do action" method.

Gun is derived from tool. Gun is instance of specific gun-type. The Gun-type is a simple data record for technical specification of the gun. It carries information about range, damage specification or standard bullet capacity. Gun, the instance, has information about charged ammo and of course its gun-type. The action of the gun is to shoot. The gun reads position and azimuth of its holder and puts bullets in the space. How to represent a bullet? First option is to simulate actual bullet object, small piece of metal flying through the town, test its collisions and travelled distance and decide its fate. Problems of this approach are mainly accuracy of the simulation and computer performance. Accuracy: do not forget that we have episodic model of the entire game. Every tick the bullet will move discreetly. What if the hit object is narrower than the one-tick bullet move distance? We will not get to know that the bullet had to hit it. Automatic weapons evoke the performance problem. Well, two doses and we have too many bullets to handle. Better solution for bullet simulation is to assume that the bullet flight is one episode moment and simulate the trajectory by one object. The impact can be calculated in one moment and the object can be shown for example for few milliseconds.

When we put one single object instead of the whole bullet flight we will get a set of objects the bullet goes through (by collision detection). We must decide which one will stop the bullet and which will be affected. Our decision is that the first solid objects in the way will be affected and stops the bullet. How to choose the right one from the colliding set? We have no information about bullet intersections with the objects. We only know that it is not null. We will use a very generic technique to deal with this issue. We can simply make an object (quadrangle) that will simulate the flight of the bullet to a specific distance. And this object we can test for collisions. To find the first hit object we use the bisection method. We start with the gun's range as length of the simulating object (bullet's flying range) and take the set of colliders that we got from space partitioning collision system. Now we recursively truncate the length and search for the end of the flight until only one collider will remain. This is the first hit object.

Second interactive content are active-objects. These are special objects with the ability to do something based on human[3] impulse. Active-objects are defined by description of an action they can do and distance from which it can be started. The active-object checks for humans in its neighbourhood and sends them information about action availability. Based on this notification humans can start and then end the action. Start of the action and its end are separated because we want to consider actions with duration in our game.

The only implemented action-object derivate is the quarter flag. We need it as partial objective for capturing the quarter. Flag has simple meaning. Player's task is to hang his flag on the pole. It takes more than one moment. The flag measures time between human starts and stops the action and if it is enough, it notifies the quarter about ownership change. We also draw a progress bar on the screen during this action.

2.5. People

2.5.1. Object Human

Game-object human has already been mentioned. We have prepared interactive content for humans. Now it is time to implement the humans. Human is a much more complicated game-object than those previously described. As it was

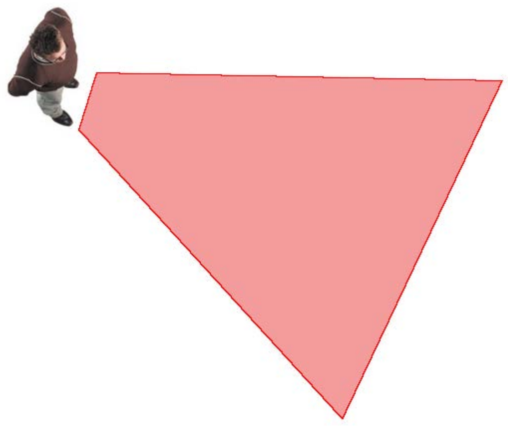
[3] Human is one of the most complicated game-object derivations. It can move, use tools, etc. In chapter 2.4 it is presented as an interactive content user. Fully it is described in its own chapter (2.5).

written above, human is a tool holder. He can perform tool action and make the action-object do its action. The next information carried by human is the list of his enemies. They can naturally be only humans. Finally we cannot forget his health state information.

2.5.2. Reflexes

Let’s go straight to the human behaviour. We need the human to perform human acting in our game. We put the main effort into every update logic. Every moment the human must decide what to do.

The first things we need to consider are reflexes. Reflexes have the highest priority for deciding what to do in this one moment. We will program two of them: balk reflex and shot reflex. As a help in reflex implementation we prepare view cone (illustrated in Picture 14). View cone is a special quadrangle that can be calculated from human’s position and specified view distance. View cone collides with objects seen by the human.



Picture 14 Human and his view cone

If the human sees his enemy, the way is clear and he is in striking distance, he shoots. If the way is not clear or the enemy is too far from the human, he makes a move toward the enemy. That is the shot reflex.

If something appears inside the view cone, human must go around it. We use a simple step aside. For this reflex quiet small view cone is sufficient. We cannot forget that human can make only one move per episode, so if any reflex moves with the human figure, other reflexes or other move-decisions are forbidden in the same update.

2.5.3. Tasks

Next move-decision after the reflexes is scheduled moving – tasks. Human has a queue of tasks that he needs to accomplish. Actually we implement it as a linked list with pointers to the first and to the last item. In future we will need to add a task with the highest priority – add it to the top of the list.

Task is an abstraction for human to act his role in the game. The task has two basic features. It and only it determines whether it is completed and the second ability is leadership. Task can lead its holder to its goal. Concrete implementations of these functionalities depend on the type of the task. We implement several types in our game. The basic type is the move-task. It finds the nearest path-graph[4] vertices for the starting position and the destination and then searches for the shortest path through the town between those two vertices. After these calculations the task leads the holder through the waypoints until he reaches the destination. Then is the task completed. This navigating mechanism is used by other task types too. Superstructure of the move-task is infinity move-task. It collects several move-tasks and repeats them in infinite loop. This type of task never says that it is complete. We usually give the infinity move-tasks to quarter guards or walkers. The quarter will look more interesting, if there is some movement. Next type of task is kill-task. Every tick the holder is navigated toward the target and if he is in striking distance, he shoots. Actually the shooting is already solved by holder's reflex, so the kill-task is technically only move-task with dynamic destination. Kill-task is complete when the target dies. We also implement action object task. This type navigates the holder to specified action-object and then makes him perform the action-object's action. After that is the task complete. The last type of task is special. Temporary task is a container for another task and it also contains a validity predicate. Every update it performs update of its inner task. In every request about completeness the predicate is evaluated and if it is not true anymore, the task returns "complete". Otherwise it returns the result of request from the inner task. The temporary task concept allows us to perform a task only as soon as some condition is true.

Back to the human update process. Human picks the top task from the list and checks whether it is complete. If it is, the task is discarded and the next task is

[4] The path-graph or the town path-graph is structure built in chapter 2.3 for path-finding needs. Do not confuse it with the town-graph which only describes which quarters are connected together.

taken instead. Now it is time to call update process of the selected task. It will move with the human and our behaviour stuffed in the human update process is at the end.

2.5.4. Post-reflexes

All that remains now are post reflexes. Post reflexes are actions without direct behaviour impact. They are only logic computations. At first, human cannot stand his enemies in the same quarter. So if there are enemies in the same quarter, human gets new temporary task to kill them with the highest priority. This temporary task will be valid until they are still in the same quarter. This post reflex is suppressed if the human has already kill-task or temporary kill-task to do in this quarter. We need this in case of the situation when the human is reaching his target and one of his enemies enters the same quarter. The human does not have to leave his target – he would put his own life in danger.

We need just one more post reflex. After human moves it is necessary to check collisions in the quarter. If he hits a box, he takes it. If he hits a building, he goes back and so on.

2.6. Opponent

The opponent[5] is a specialised extension of human. We use the same logic for reflexes, task solving and post reflexes. What we put as an extra for opponent is task planning. Opponent plans his tasks to win the game and then he acts like an ordinary human.

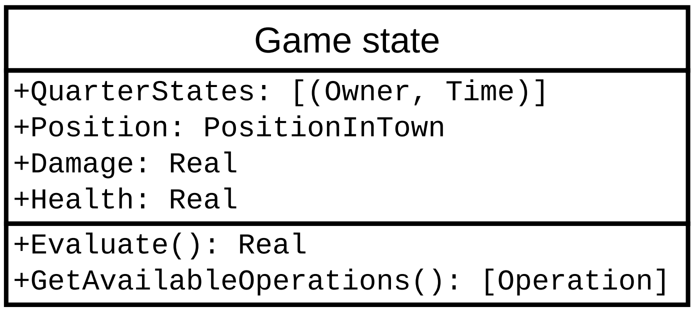
We add only two post reflexes to the opponent. First is the actual task planning which is needed if the opponent has empty task's list or after timeout elapsed. Plan needs revisions during time because opponent is situated in a stochastic space – the planner does not consider other humans behaviour and does not mainly know what the player will do. Second added post reflex is flag checking. Like an ordinary human checks for enemies in his quarter the opponent checks if he can capture the flag in the quarter he is located in. Of course we add this action object task to the opponent only if he already has no other task in this quarter.

[5] The opponent is a term for software component which represents the AI player.

2.6.1. Task Planning

The most interesting thing in opponent's program is the task planner. Because of stochastic space we cannot have optimal plans and we need to plan tasks during the game play, so we need to do this fast. Due to these factors we choose forward planning and we will generate only partial (short) plans – for the near future.

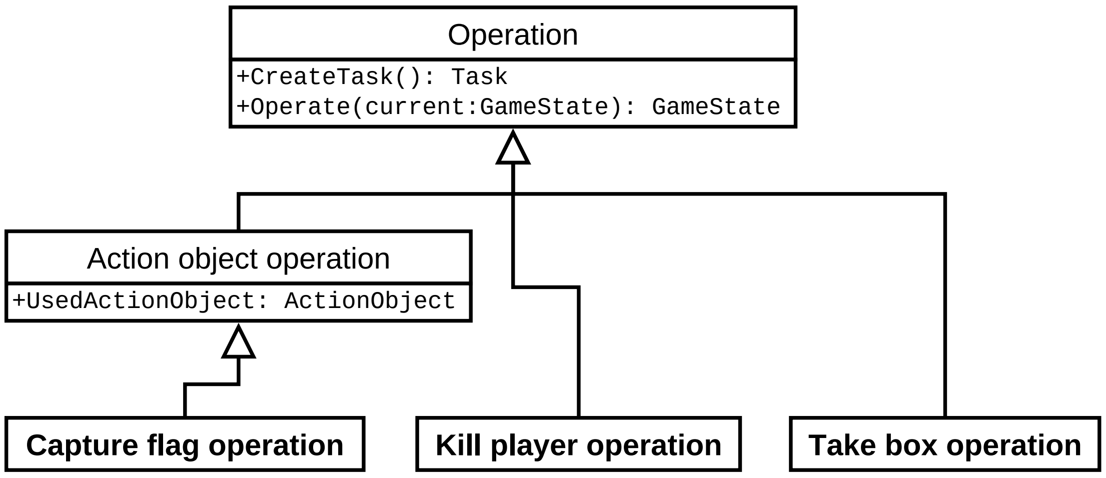
For planning – graph searching we need game states which are represented by vertices of searched graph and operations as edges. The game state contains description of whole city seen by the opponent at one moment. It has information about quarter ownership – which quarters are owned by whom and for how long, opponent's position, his health and about his potential damage ability (see description of game state in Picture 15). Every single state must be evaluable. We prepare a procedure that converts a given state into a number indicating its quality for the opponent. In the evaluation the opponent's quarters are good and naturally the quarters owned by the player are bad. From the length of the time for which a person holds the quarter we calculate the number of guards in the quarter and multiply it by the quarter quality index. Health and damage are included in the state quality calculation too.



```
Game state
+QuarterStates: [(Owner, Time)]
+Position: PositionInTown
+Damage: Real
+Health: Real
+Evaluate(): Real
+GetAvailableOperations(): [Operation]
```

Picture 15 Game state structure schema

Transitions between states in planed simulation are controlled by operations. Operation is a procedure that simulates accomplishment of some task and changes the input game state into the assumed output state. Operation types are based on task types. We implement action operation; especially flag capture operation, operation for killing the player and take box operation (operation types diagram is in Picture 16). Every operation calculates time that is needed for it and prepares game state, like it can be after computed time elapses. Then it adds its own specific impact – for example turns quarter ownership into opponent's or grows opponent's damage ability. If we add some new task types, action objects or tool types in the future, it is necessary to take account here and add appropriate operations. Without operations the opponent will not use the new content that we have added.



Picture 16 Operation types schema

We must add a procedure that will return available actions for a specified game state. Here we put causal conditions. Without potential damage it is not possible to go kill somebody, and so on.

Now we have everything for a planning process. Forward planning is simple graph searching [4]. We start at the current game state and after considering all available operations we search recursively the new created states. We are looking for a state with the best quality. Potentially the best state is the one with all the quarters owned by the opponent, but this searching has to be as fast as possible. We said that partial plan will be sufficient. How to search for only partial plan? We specify constant length of the partial plan and search for the best with this length (depth in the searched tree), but this would be still too slow. The number of available operations is the branching factor of the search tree and it's always greater or equal to the number of free quarters. For example when we search for a plan of length 5 and there will be 9 free quarters, health and tool boxes to take and ability of killing the player, the search tree has potentially $5^{12} = 244140625$ nodes. It is definitely not possible to do this operation in a regular update process of the opponent. After all it is not necessary to have results from planning in the same update process, in which it started. Thus we run the planner asynchronously. Task planning is a separated operation, so it is not necessary to implement a lot of synchronisation primitives. The only part that shares memory with the main thread of our game is saving the plan into the task list. Thus using the task list is the only one part of the opponent implementation which needs synchronisation.

2.7. Rendering and Playing

2.7.1. Player

Player[6] game-object is, like the opponent, derivate of human. We reuse mechanisms like holding and using tools, control action objects, and so on. What we definitely suppress is the update procedure. The entire human's behaviour is not desired here. Player's acting is controlled only by the user of our software. Actually it is much easier to implement player's update process then the human's. All we need to do is check for pressing any of keys that are set as game control and based on the caught ones (detected pressed keys) call relevant behaviour from human's part of the player's code. For example pressed key W moves the player one step forward – the "step forward" procedure is already defined in the human's implementation. We also calculate difference of mouse cursor position and rotate the player.

2.7.2. Rendering

Since we have defined the world our game is situated in, we have not answered the question about drawing it on the screen. It would be easy if we had an ordinary vector space. We would transform every object by set of matrices by its position and rotation then by view and projection matrix and render it on the screen. When we want to calculate absolute position of an object from another quarter by transformation of its position according to quarter interfaces connection, we find out that the object has more than one possible result position. One for each walk through the town graph from camera's quarter to the object's quarter. This is not the right way to do the drawing. Next reason why we should not draw all the objects in the town is the software performance.

First we can draw the quarter where player is located. There is no possibility of doing it wrong. And the remaining quarters? In the quarter generating process we determined that the quarter is bordered by buildings. There is practically no view of other quarters except places near the interfaces. So we make a decision that only one neighbour quarter will be drawn. We simply choose the nearest interface in our quarter and draw the quarter from opposite interface. If the interface streets are long

[6] In this chapter the player (in sources class Player) is term for special game-object that transmits the user's control into the game process.

enough, this method works fine. We just must not forget that the opposite quarter has to be drawn transformed so that it fits together with our quarter.

2.8. Settings, Xml Configurations and Menus

It is a frequently used fashion to prepare some opened parameters in software product that can be set by user. Most of the games use graphical user interface (GUI) made right inside the game. They are using the game’s uniform graphical design and are in full screen mode. We decided that for us classical windows will be enough. We use windows forms and controls for game menu and settings. This is uniform with the whole operating system environment.

Before starting the game we show the user windows with settings tabs and a button for the game beginning. In the video settings tab we let him choose screen resolution and set whether he wants to run our game in full screen mode. Controls tab should contain mouse sensitivity track bars, possible choose of mouse inversion and settings of control keys. In the game tab we let the user set the number of quarters in the town. We must think about our implementation of town generation and set minimal and maximal value to this option because of the performance.

Next level of configuration is “modding”. These are changes in the game that do not require code modifications and new compilation. We support it by adding xml configuration files. We prepare files with gun types. They will describe types of guns with all their properties that are used in the gun implementation. Second xml will determine used content like 3D models or sounds. So a more experienced user can change what he will see or hear during the game. “The modder” should know that he does these changes only at his own risk. Of course we must make our own versions of these files and add to our game some default content, because without it the game would be not able to run at all.

3. User Documentation

3.1. System Requirements

To run this software you need an IA-32 (x86) or IA-32e (AMD64) computer. Minimal CPU frequency is 2.2GHz and you should have at least 2GB RAM installed. Dual-core processor is recommended. You need a 3D graphic accelerator supported by Microsoft DirectX and having at least 256MB of graphic RAM. We support the Microsoft Windows XP or higher operating system with .Net framework 4.5 or newer and Microsoft DirectX installed. This software does not support any UNIX operating system with Mono framework at all.

3.2. Installation

There is an Install folder on the attached CD. Look for the setup.exe file, which is situated in this folder. Running the setup.exe will show you the installation wizard. It is a standard .Net tool used by many other applications. Your computer will be checked for the needed libraries and if some of them are missing, the wizard will offer you to download the right versions. Then you will go through a fast installation of this game. At the end you will get a shortcut in your Start > All Programs menu.

You can uninstall this software using the common Windows practice: remove it in Programs and Features wizard located in Control Panel.

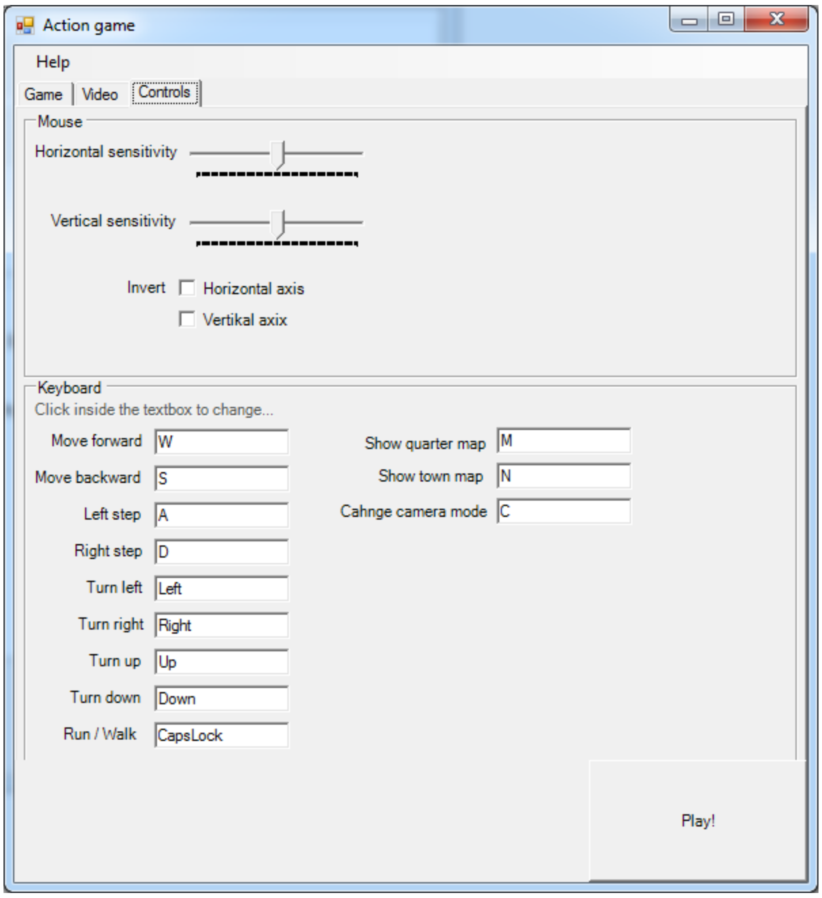
3.3. Start and Settings

First thing, that will show after you run the software is the Main menu window. Here you can set up various options. The Main menu window is divided into three tabs: Game, Video, Controls. These tabs represent groups of the options by their meaning. In the Game tab you can set:

- Number of quarters in the town
- Number of first-aid kits inside one quarter
- Number of gun-boxes inside one quarter at the beginning of the game

The Video tab allows you to decide if the game will run inside a window or if it will be in full-screen mode. Here you can also specify graphic resolution. The resolution can be selected from the table of prepared values in the combo-box or you can easily write your own special value into it. You only need to fill the field according to the pattern (WIDTHxHEIGHT).

If you want to set the controls of the game, go to the Controls tab. The Controls tab is splitted into the Mouse part and the Keyboard part. Like in any other 3D game you can set sensitivity of the mouse and invert both axes. Mouse buttons have their functionalities fixed: left button shoots; the wheel’s rotation changes the selected gun and the right button performs a secondary action if it is available. Next you can specify what keys you want to use to control the game (for list of the controls you can see Picture 17).


Action game

Help

Game | Video | Controls

Mouse

Horizontal sensitivity

Vertical sensitivity

Invert ☐ Horizontal axis

☐ Vertikal axix

Keyboard

Click inside the textbox to change...

Move forward	W	Show quarter map	M
Move backward	S	Show town map	N
Left step	A	Cahnge camera mode	C
Right step	D		
Turn left	Left		
Turn right	Right		
Turn up	Up		
Turn down	Down		
Run / Walk	CapsLock		

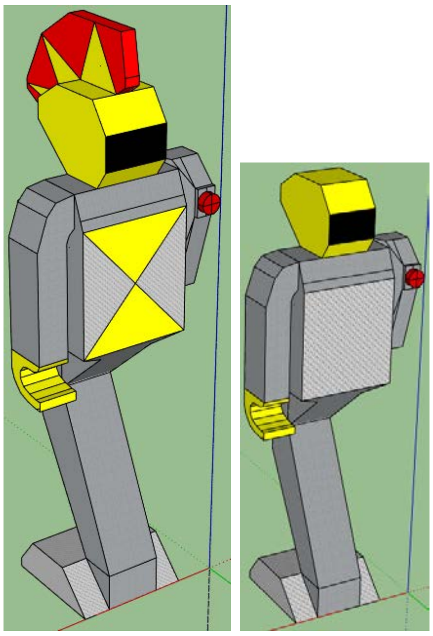
Picture 17 The main menu window with Controls tab selected

There is one more keyboard control that cannot be set in the main menu window. It is a cheat for „God-mode“. When you swich into the God-mode, you are imortal. The magic key combination is to press left Ctrl and left Alt keys simultaneously. This is only for the purpose of testing. Do not abuse it for the ordinary gaming.

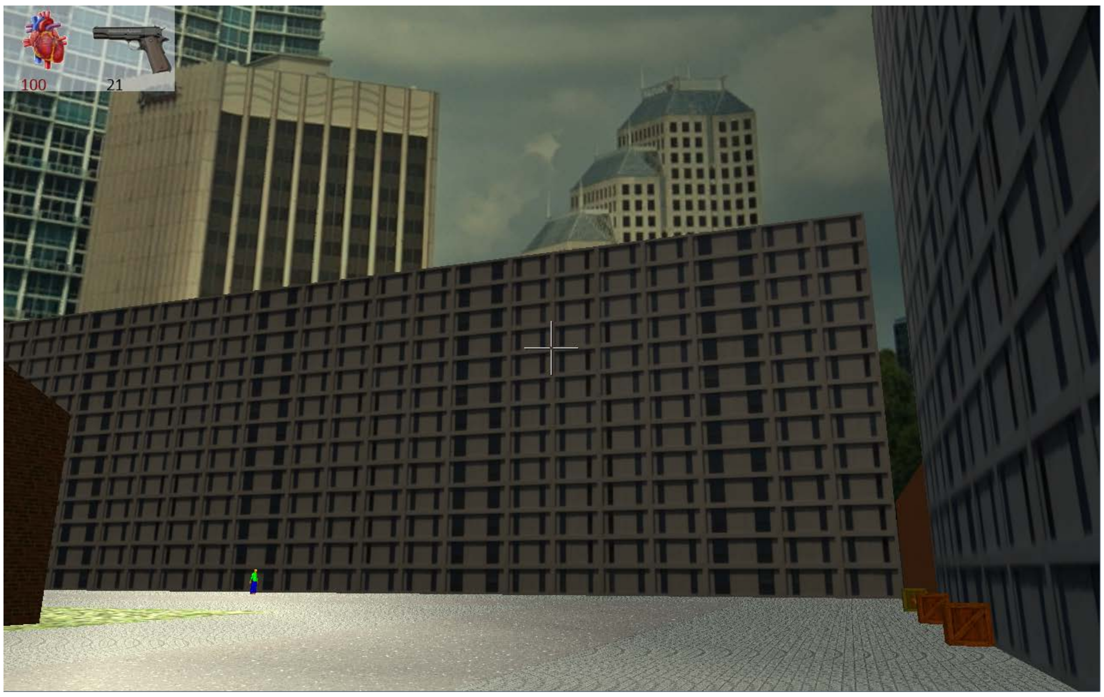
After all the settings you wanted are done, press the Play! button and wait for the game to prepare the town.

3.4. Gameplay

At the beginning of the game you are located somewhere inside the only one quarter you own. There are no guards at this moment. They will appear during the time – it is like you have just captured this quarter. Both of the players (you and your opponent) have their property marked by their personal colour. Your colour is blue and the opponent has yellow. Both of the players have parts of their bodies coloured with their personal colour. Similar are the robots that are doing the guarding for you (example of opponent’s colour used on the robot’s body is in Picture 18).



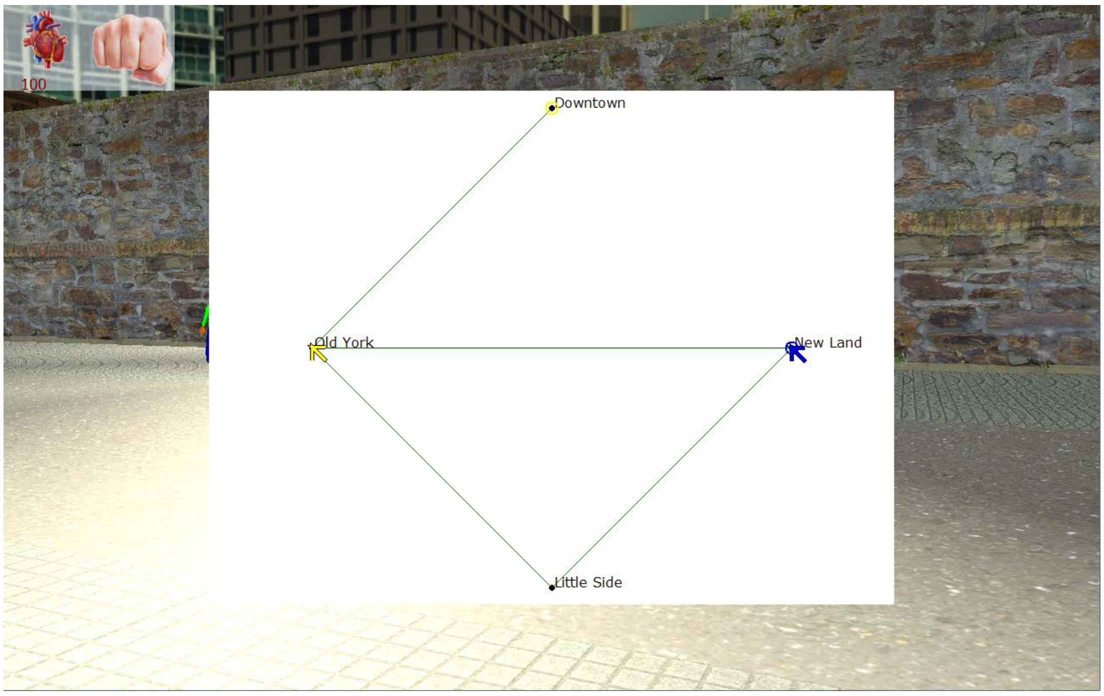
Picture 18 Your opponent and his quarter guard



Picture 19 Gameplay and boxes on the sidewalk

Except the view of the town you can also see your status panel and the targeting cross on the screen during the gameplay (in Picture 19). The status panel is always located in the upper left corner and it tells you your percentage of health. It also informs you about the currently selected gun and available ammo.

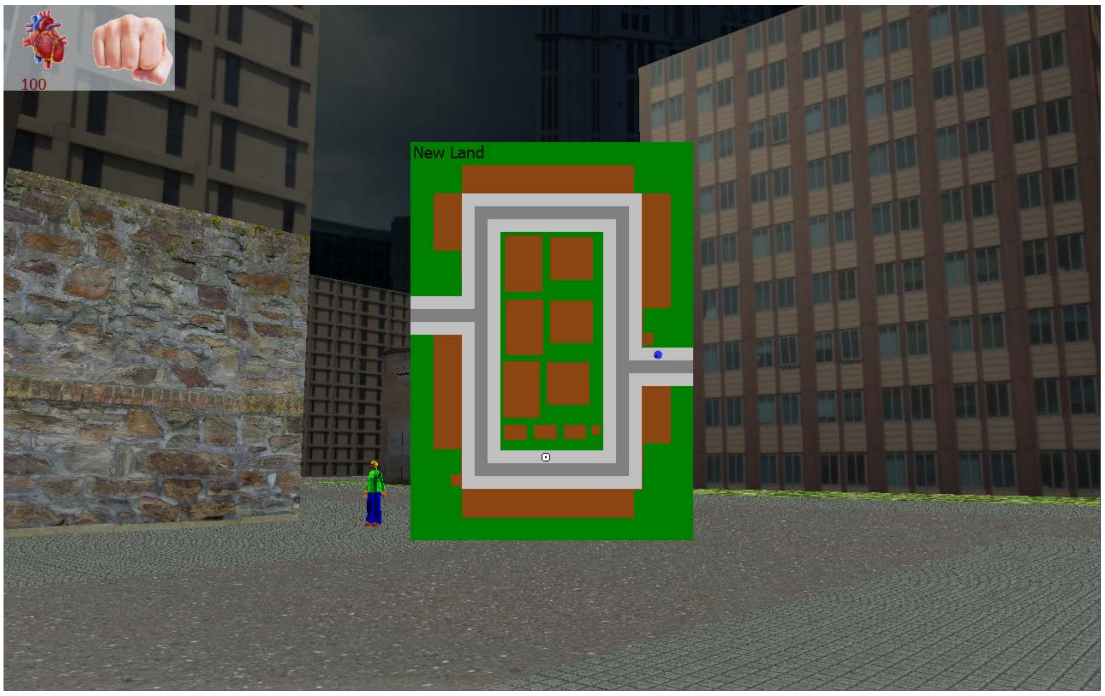
First of all you should check the schematic map of the town you are playing in. Press and hold the town-map key (by default N) for showing the town map. Here you can see which quarters are connected together, which are owned by whom and in which quarter are you or your opponent located. This figure (visible in Picture 20) will disappear after releasing the town-map key.



Picture 20 Town-map shown during the gameplay

On the town map the town-graph is shown. The quarters are represented by black coloured vertices. The quarters connected together have a green edge between them. If the quarter is owned by the player or his opponent, the corresponding vertex is inside an owner's color circle (blue or yellow). There is also always an arrowed sign that shows in what quarter is currently the player and his opponent located. If they are in the same quarter, there will be only one arrow (the second is beneath).

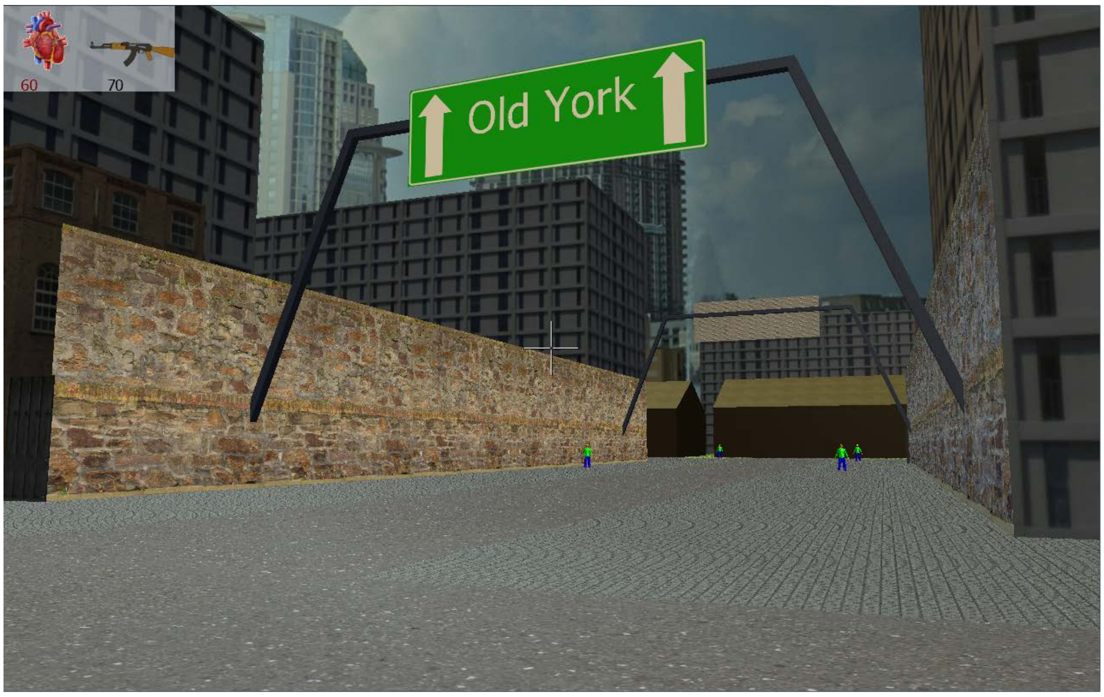
Second gadget that can help you to get oriented is the current quarter map. You can see it by pressing the quarter-map key (by default M). This is not only a graph schema, but it is a real map (visualised in Picture 21). The map shows you the roads, sidewalks, buildings, the quarter flag and the location of the player.



Picture 21 Quarter-map shown during the gameplay

Everything on the map is drawn on a green background. The green color means that there is only a grass and nothing else or you cannot even get there. Brown coloured are buildings and walls, which means that you cannot go through. The shades of grey stay for the roads and for the sidewalks. The most important things, this map shows, are positions of you (naturally the blue point) and of the quarter's flag (white circle with black dot inside).

There is one more thing you will certainly appreciate. The roads that are connecting two quarters togehter are labeled by roadsigns (like in Picture 22). The roadsign tells you where you are going using this road and furthermore in whose property the opposite quarter is. The roadsign changes its background color according to the corresponding quarter owner. Unowned quarters have green roadsigns.



Picture 22 The navigating roadsigns

Now it is time to play the game already. Every time you need to make a decision whether you are going to capture another quarter or rather go to kill your opponent. You should also check your status panel and consider searching for a gun or a healing box (you can see the boxes in Picture 19). Always watch your opponent! He can show up any time and shoot you in the back. He does not wait for anything, so while you are discovering the town, he is on his way to capture the next quarter or he is preparing to kill you. Do not worry; you will be informed about his activities by dialog messages on the top of the screen.

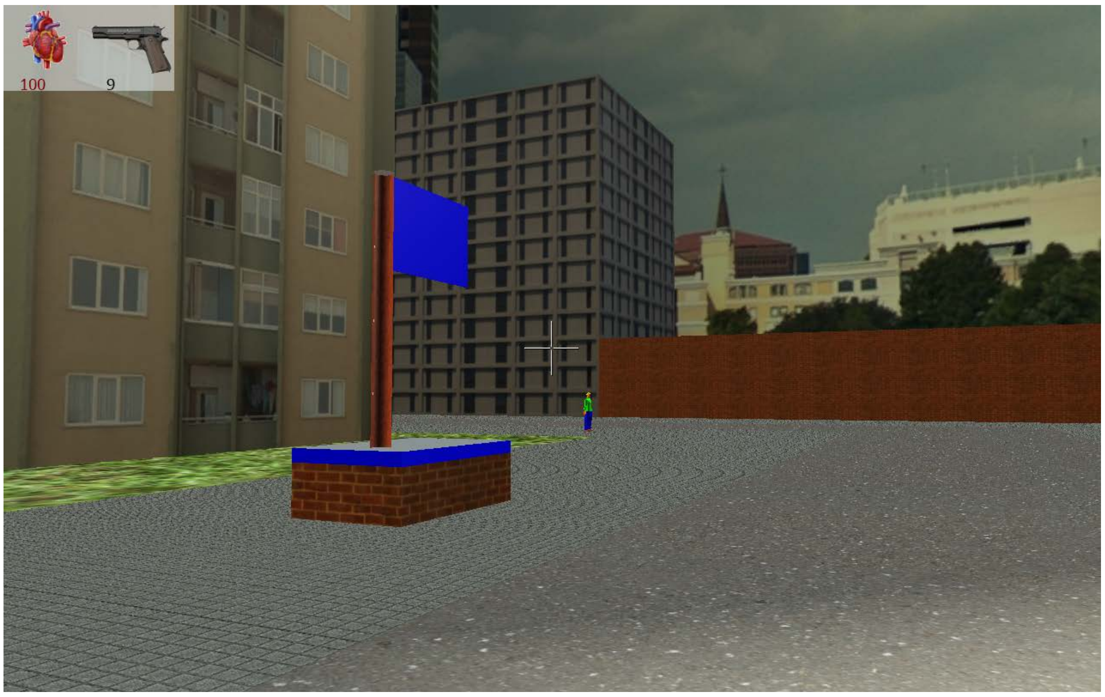
There are five types of guns defined in the basic gun set in the game:

Table 1 Available gun types and parameters

Type	Damage [%]	Range [m]	Shoot timeout [ms]	Needs ammo
Fists	5	0.5	800	No
Pistol	40	100	600	Yes
AK47	30	80	130	Yes
Sniper rifle	100	300	1400	Yes

When you come very close to a quarter flag available for capturing (it's not yours), an action indicator will show up. The action indicator is displayed in the lower left corner as a red circle with white exclamation mark inside. This means that you can perform a secondary action – it is capturing the flag in this case. Press and hold the right mouse button until a blue progress-bar will reach 100%

of the width. Then you can release the button. Congratulations, you have now captured the quarter (captured flag in Picture 23).



Picture 23 The flag inside the quarter owned by the player

You can watch a yellow progress-bar while your opponent is capturing a flag too. This means that you can quickly react and capture it for yourself before the guards will show up. Moreover the dialog messages inform you precisely about just captured quarters.

3.5. End of the Game

The game ends when all the quarters have the same owner. The game will close and a window, telling you who won, will show up. After that you can exit the application completely or configure a new game and play again.

Conclusion

This thesis has brought in a new computer game idea. This game may seem crazy or confusing to somebody, but that is why it is so interesting. We came up with solutions based on known techniques transformed for our purposes. We took the reader through the real-time game programming mechanisms in general and described the whole implementation of our idea. We introduced the software architecture of the bizzare surrounding and the behaviour of artificial beings inside it. The result software is a usable game that most likely cannot be compared with any top game titles, but it is original and fun. Other benefits of this work can be the possibility of configuration and source codes prepared and documented for future development.

There are lots of areas that we can extend, append or modify. For example the game would deserve better and more sensational content – 3D objects, sounds, animations and so on. Then the rendering part should be more sophisticated: using advanced lights, visual effects – use all the advantages that nowadays GPUs have. Another way to improve this game is to add more interactive content. Define more guns or extend the tools and action-objects concepts – to add other types of tools or action-objects. The next further development can for example add a multiplayer mode. Either only network game for more players, or more and better AIs in a cooperation or death-match mode. Finally, the most interesting improvement could be an extension of the bizzare city definition. Our space is theoretically equivalent to an orientable arbitrary surface. Our map is in fact a polygonal representation of such surface. So the natural extension could be supporting non-orientable surfaces too. That would create special interfaces between the quarters which would mirror your view. Your left hand would be simply on the right side and conversely. This would be definitely fun.

Attachments

The attached CD contains:

- Electronic version of this text
- Documented source codes of our game
- The game installation files

Bibliography

[1] Microsoft Developer Network: XNA Game Studio, [2013-03-09] at http://msdn.microsoft.com/en-us/library/bb401006.aspx

[2] Josef Pelikán: Datové struktury pro prostorové vyhledávání, [2013-05-16] at http://cgg.mff.cuni.cz/~pepca/lectures/pdf/2d-06-datasurvey.pdf

[3] Stuart J. Russell and Peter Norvig, 2010, Artificial Intelligence: The modern approach, 3rd ed. Upper Saddle River, New Jersey: Prentice Hall, ISBN-13: 978-0-13-604259-4

[4] Malik Ghallab, Dana Nau and Paolo Traverso, 2004, Automated Planning Theory and Practice. San Francisco: Morgan Kaufmann, ISBN-13: 978-1-55860-856-6

[5] Games++ forum: Collision detection Vertex-in-triangle check, [2013-04-02] at http://www.gamespp.com/algorithms/CollisionDetectionTutorial.html

[6] Valve Software: Portal (Electronic Arts, Steam, 2007), [2013-05-16] at http://www.valvesoftware.com/games/portal.html

[7] The Bitmap Brothers: Z (Eon Digital Entertainment, 1996), [2013-05-16] at http://www.bitmap-brothers.co.uk/our-games/past/z.htm

[8] Rockstar North: GTA: San Andreas (Take-Two Interactive, 2004), [2013-05-16] at http://www.rockstargames.com/sanandreas/

[9] International Telecommunication Union: Recommendation ITU-R BT.470-6 [2013-05-18] at http://www.itu.int/dms_pubrec/itu-r/rec/bt/R-REC-BT.470-6-199811-S!!PDF-E.pdf

List of Pictures

List of Tables