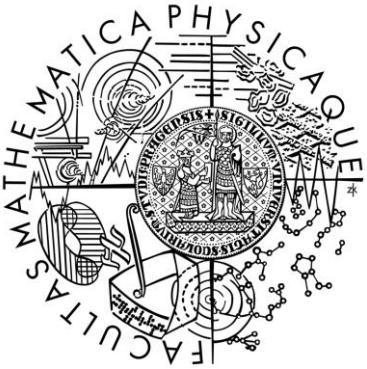
Charles University in Prague

Faculty of Mathematics and Physics

BACHELOR THESIS



Štěpán Havránek

3D action game in a bizzare city

Department of Theoretical Computer Science and Mathematical Logic

Supervisor of the bachelor thesis: Mgr. Tomáš Balyo

Study programme: Computer Science (B1801)

Specialization: Programování Bc. R4 (NIPR4B)

Prague 2013

[Sample: Bound Sheet – a signed copy of the "bachelor thesis assignment". **This assignment is NOT a part of the electronic version of the thesis. DO NOT SCAN**.]

[Sample: Here you may thank whoever you wish (the supervisor of the thesis, the consultant, the person who lent the software, literature, etc.)]

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 date.....
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: [abstract of 80-200 words in Czech, but not a copy of the assignment of the bachelor thesis]

Klíčová slova: [3-5 keywords in Czech]

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: [abstract of 80-200 words in English, but not a copy of the assignment of the bachelor thesis]

Keywords: [3-5 keywords in English]

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. 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

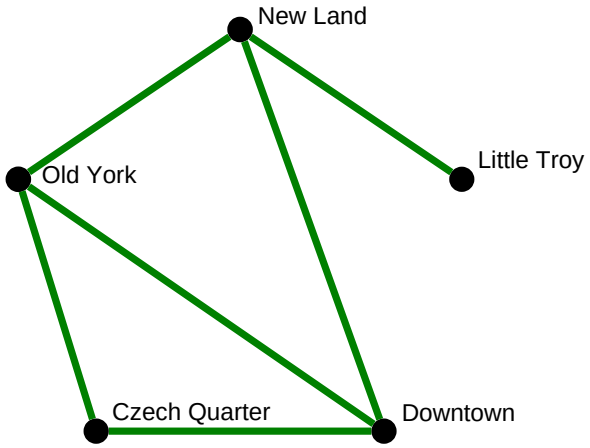
This thesis mainly describes implementation of the whole action game situated in the introduced town. We will begin with description of our software project architecture based on Microsoft XNA Game studio [1]. Then we will go through real-time programming issues, data representation, used algorithms 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.

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 continuous, but there can be a quarter with degree only 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. 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. 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. Rule is that we limit the sum of yours and enemy's guards. So if you capture 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 approach. 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. Attention, not only you can take guns from boxes. Your opponent will do it too. The ones from second and third category are simply in 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 generally killed, he will lose all of his quarters except one, if he has at least one. In his only quarter he will appear alive again. If he does not have any quarter, he will show in some empty one. Again, if you lose your quarters by getting shot, your guards will stay there. Only new ones do not appear anymore.

1.2. Similar games

O tehle hrach by tu mela byt zminka – neni to jako vysledna kapitola…

Portal

Z

San Andreas

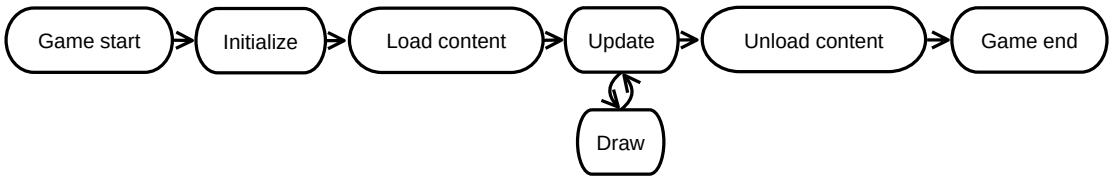
2. Implementation

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. Frequency 25Hz is generally well-known frame-rate value that human eye perceives as continuous

moving. For example the European standard for television broadcasting uses this frequency. 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. XNA libraries provide prepared process model for the whole game. First we need to initialize our components, then load all needed content. Content loading can be very slow operation 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.



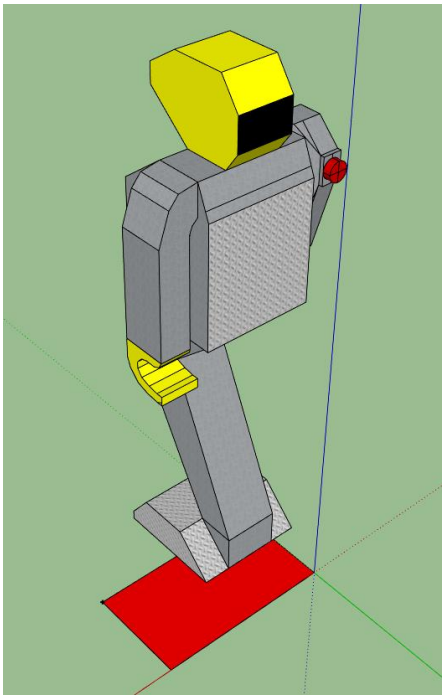
Picture 2 Game life cycle diagram

Good idea is to have this process distributed into separate components. A component model benefits clarity of the whole solution. We have several smaller modules running according the diagram metioned in Picture 2: 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

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 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. The quadrangle is representing the object's floor projection (ground-plan). These quadrangles are used for collision detection between objects.



Picture 3 Robot and its quadrangle projection (red)

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 catch 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 collision too. Since we know that borders off these

[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.

triangles do not in 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 the all triangle vertices by vectors. Then we calculate angels between all the vector pairs. The tested point is inside the triangle if and only if sum of the angles equals to two PIs [5]. Moreover if we do not found the first triangle inside the second, we must test whether the second is inside the first.

Considering the game logic, the use of quadrangles is not the best way to represent base of objects in the game. Quadrangle is defined by four points and it can be little bit confusing if we imagine that we have prepared 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 position of the base four corners. Better way is to add next level into space hierarchy. We present the game-object[2]. Game-object is 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 axes of three-dimensional space. Bottom base of this cuboid is rectangle. And this rectangle is represented by game object. Game object 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. Game object, as the name says, describes 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. Position of game-object is now specification of the concrete part of the world

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

and 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.

Back to the game objects once again. 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-object, or flat ground objects and plates for work with only texture instead the 3D-object.

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. 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 position defined by a pair of vector and quarter.

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.

There are many objects we need to test for collision against each other in the town quarter. It is not very practical. Collision detection is routine which has to be run during every update. And it is necessary to do that. 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.

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

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 used data structures like BSP Trees or Quadrant Trees 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. Leafs of the search tree contain objects that are located in the area specified by all the nodes above the leaf in the tree. Advantage of this data structure is that leafs must not

be in the same depth. When you need to test collision, you know that in collision can be only objects from the same leaf. Because two objects from different leaf are not in the same part of the space. However moving objects create problem 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 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. It is good how quick the space partitioning tree test for collisions is, but we want to search for the right area for an object in constant time.

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. No trees, no logarithms in complexity. We create parts of fixed size formed into squared grid. The part the specified object belongs to is simply calculated as position in quarter divided by grid size. Similar partitioning would be result of Quadrant Tree with evenly distributed objects. So when we try to generate the world by uniform pattern, method of grid partitioning will not be worse by allocated memory than partitioning trees. Now with our data structure and evenly distributes 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 flying bullet kills and removes 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.

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.

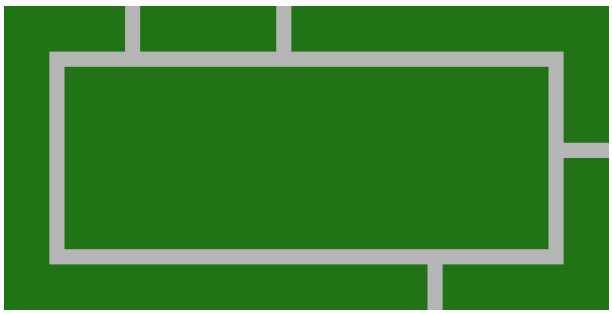
First comes the input. And the only input is the number of quarters (n) that will be in our town. We prepare empty non-oriented graph with n vertices. Into this

graph we add edges. Because we want to have this graph continuous (one component), we insert 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 cycle, so we don't do a cycle – only simple path of length n. Result graph what we have could be a regular output, but the game with always repeating 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 random number generator to decide whether add the edge to the graph or not. Now we have graph describing our town. The vertices represent quarters and the edges are joining streets between them.

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 road and sidewalk network. Every segment of the road is lined with sidewalk. We start with "border" road of the quarter – smaller rectangle around the quarter formed by road and sidewalk.

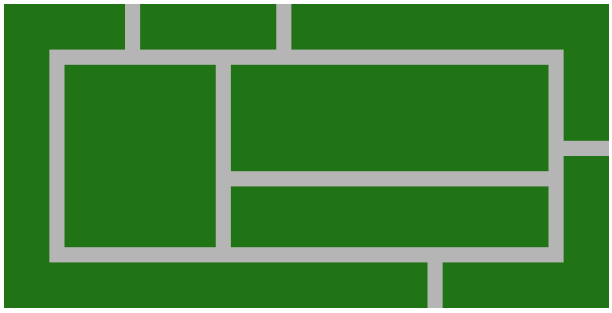
Picture 4 Town quarter with border streets

To the border road we connect interface roads.



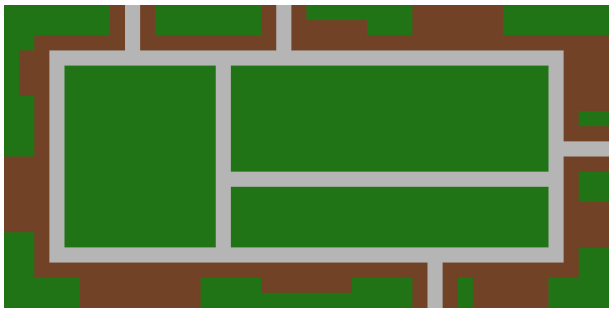
Picture 5 Town quarter with interface roads (quarter with degree 4)

Inside the border road rectangle we have an empty space. Using 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 road network done.



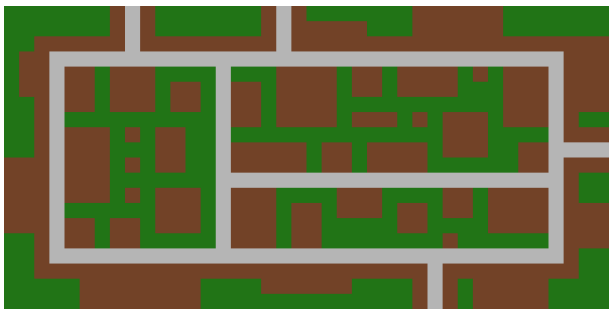
Picture 6 Town quarter with inner road network

Since we have roads inside the quarter we can start putting in buildings and decoration objects. At first we build border buildings, fences and walls. It is necessary to prevent the player get 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.



Picture 7 Town quarter with border buildings, fences and walls

Than we have empty rectangles inside the road network. These we fill by buildings with spaces between them. Or we do not. We use the random number generator to decide whether fill the empty space by buildings.

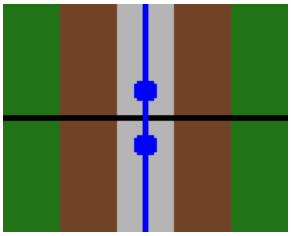


Picture 8 Complete town quarter with inner buildings

We have done all the quarters and its interfaces are joined – every interface has 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 continuous so we have to go somewhere thru 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 search the shortest path between them. That is why we implement classical A* algorithm [3]. We need good heuristic for this graph search algorithm. Because our graph generally has cycles, we need the heuristic to be not only admissible but it has to be monotonic too. A* algorithm commonly uses distance in Euclid metrics. However our space does not even theoretically satisfy the triangle inequality axiom. Euclid metrics we can use 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. We have chosen it in the town generation process. This heuristic is monotonic. When the target will be in another quarter, our heuristic is always lower or equal then the real price of the path-graph edge into next vertex (which is in another quarter) and heuristic there is always non-negative. To get to another quarter we must go through an interface.



Picture 9 Two opposite quarter interfaces with the path-graph vertices connected by edge of constant length

2.4. Boxes, tools and action-objects

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

We will start with tools concept. Tools will be a generic entity in our game that human can handle. At first we need boxes. Boxes will be first special objects in the game. 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 human, who takes it, will turn into hundred percent healthy. Toolbox contains variation of tools.

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

Gun is derived from tool. Gun is instance of specific gun-type. The Gun-type is 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. Action of the gun is to shoot. The gun reads position and azimuth of its holder and puts bullet 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 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 very generic technique to deal with this issue. We can simple make object (quadrangle) what will simulate flight of the bullet to specific distance. And this simulating 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 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 progress bar on the screen during this action.

2.5. People, reflexes and tasks

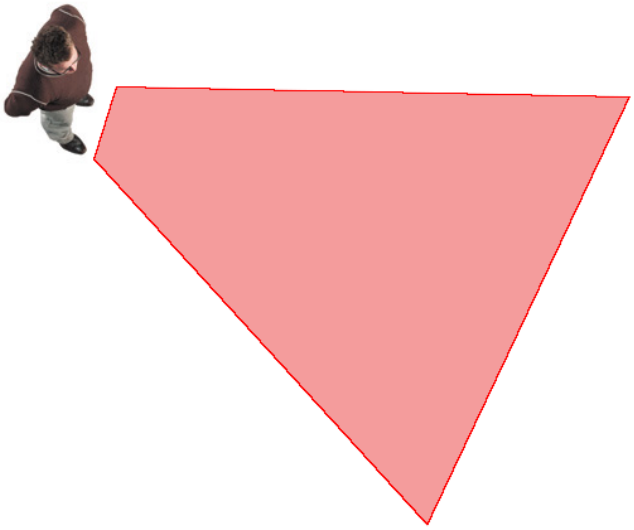
Game-object human has already been mentioned. We have prepared interactive content for humans. Now it is time to implement the humanity. Human is much more complicated game-object than those previously described. As it was written above, human is a tool holder. He can perform tool action and make the action-object do its action. The next information carried by human is list of his enemies. They can naturally be only humans. Finally we cannot forget his health state information.

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 the 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:

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

balk reflex and shot reflex. As a help in reflex implementation we prepare view cone. View cone is special quadrangle that can be calculated from human's position and specified view distance. View cone collides with objects seen by the human.



Picture 10 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 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 move-decisions are forbidden in the same update.

Next thing after the reflexes is planned moving – tasks. Human has queue of tasks that he needs to accomplish. Actually we implement it as linked list with pointers to the first and to the last item. In future we will need to add 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 type of task. We implement several types in our game. The basic type is the move-task. It finds the nearest path-graph[4] vertices for

[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.

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 implement also 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 container for another task and it also contains validity predicate. Every update it performs update of its inner task. Every request about completeness the predicate is evaluated and if it is not true no more, the task returns message "complete". Otherwise it returns 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 so, task is discarded and the next task is 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.

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 box, he takes it. If he hits a building, he goes back and so on.

2.6. Opponent, task planning

The opponent[5] is 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 flag in the quarter he is located in. Of course we add this action object task to the opponent only if he has not already another task in this quarter.

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. Every single state must be evaluable. We prepare procedure that converts given state into number indicating his quality for the opponent. In 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 person holds the quarter we calculate 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.

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

Transitions between states in planed simulation are controlled by operations. Operation is procedure that simulates accomplishment of some task and changes input game state into 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. 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 grow 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.

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

Now we have everything for planning process. Forward planning is simple graph searching [4]. We start at current game state and after considering all available operations we search recursively the new created states. We are looking for 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. Number of available operations is branch of the searched tree and it's minimally always greater or equal than the number of free quarters. For example when we search for plan of length 5 and there will be 9 free quarters, health and tool boxes to take and ability of killing the player, the searched tree has potentially $5^{12} \cong 244140625$ nodes. It is definitely not possible to do this operation in 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 planner asynchronously. Task planning is 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 plan into the task list. Thus using the task list is the only one part of the opponent implementation which needs synchronisation.

2.7. Player, camera and game controls

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 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 behaving 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.

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, but in our case? 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 in. 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 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.

[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.

2.8. Settings, xml configurations and menus

It is 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 to user windows with settings tabs and button for the game beginning. In 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 thing about our implementation of town generation and set minimal value to this option and maximum because of performance.

Next level of configuration is "modding". These are changes in the game that does not require code modifications and new compilation. We support it by adding xml configuration files. We prepare files with gun types. It will describe types of guns with all their properties what are used in the gun implementation. Second xml will determine used content like 3D models or sounds. So more experienced user can change what he will see or hear during the game. "The modder" should know that he do 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. Installation

3.2. Start and settings

3.3. Gameplay and controls

3.4. Game ends

Bibliography

[1] http://msdn.microsoft.com/en-us/library/bb401006.aspx

[2] (Neco od Pelikana o BSP stromech)

[3] Artificial Intelligence, The modern approach (doplnit)

[4] Automated Planning: Theory and Practice (doplnit)

[5] http://www.gamespp.com/algorithms/CollisionDetectionTutorial.html