Contents

[Introduction 1](#_Toc355707718)

[1. The game 1](#_Toc355707719)

[1.1. Detail description and rules 1](#_Toc355707720)

[1.2. Similar games 3](#_Toc355707721)

[2. Implementation 3](#_Toc355707722)

[2.1. Program architecture for real-time game 3](#_Toc355707723)

[2.2. Space and the game world 4](#_Toc355707724)

[2.3. Town generator 7](#_Toc355707725)

[2.4. Boxes, tools and action objects 8](#_Toc355707726)

[2.5. People, reflexes and tasks 10](#_Toc355707727)

[2.6. Opponent, task planning 12](#_Toc355707728)

[2.7. Player, camera and game controls 14](#_Toc355707729)

[2.8. Settings, xml configurations and menus 14](#_Toc355707730)

[3. User documentation 15](#_Toc355707731)

[3.1. Installation 15](#_Toc355707732)

[3.2. Start and settings 15](#_Toc355707733)

[3.3. Gameplay and controls 15](#_Toc355707734)

[3.4. Game ends 15](#_Toc355707735)

[Bibliography 15](#_Toc355707736)

# Introduction

There are many action games with 3D graphical visualisation made. Main reason for 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 simulation of this world using 3D. Aim of this paper is different from these ideas. This work deals with a game situated in a space which does not follow basic physical laws of our world.

Let’s figure out a game that maybe looks like a classical 3D. But parts of 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 world of our game.

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

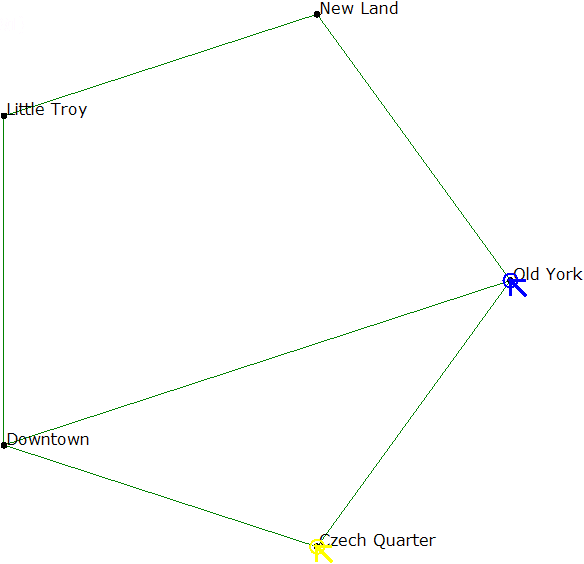
This paper mainly describes implementation of whole action game situated in the introduced space. We will begin with model of the program based on Microsoft XNA Game studio [1]. Then we will go thru real-time programming issues, data representation, used algorithms or modified versions of well-known algorithms for our specific case. The end station of programming part will be implementation of AI for player’s opponent. During this part we will point out several scopes for further development too.

User documentation for creation we make will not be omitted in second part. Reader will find out how to set up and, of course, play the game.

# The game

## 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 degree of one quarter can be only 1.



Picture Town quarters make a graph

Every quarter has somewhere inside flag or empty flagpole. This indicates who the owner of this quarter is. Your flag means that this quarter is in your property. Otherwise the quarter can belong to your opponent or to nobody. Either way good advice is to capture it. The goal of the game is to capture all the quarters in the town. When you or your opponent have reached this objective, the game is over. The game begins with one quarter owned by the player and one opponent’s. Rest of town is without an owner.

Don’t worry about the quarters you have captured. Your guards gradually appear there. They have only one thing to do. When an enemy comes into this quarter, he becomes target of the 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. Problem is that the limit for the number of guards in one quarter is for sum of all the guards – yours and opponent’s. So if you capture quarter full of enemy guards, yours would not appear until you kill at least one of the other.

How to kill somebody? You can always use your hands, but it’s not recommended approach. It’s not practical. There are lots of better gun types you can use. We define four categories of guns:

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

Guns from the first category are held by everyone at start of the game. These from the fourth category are available only in boxes. Attention, not only you can take guns from boxes. Your opponent will do it too.

Since we have guns and our enemies have guns, it is necessary to use them. Except the guards you will need to shot directly the opponent several times. Because when the opponent gets killed, he loses all of his quarters except one if he has at least one. And in the one he will appear again alive. If he does not have any quarter, he will show in some empty one. The same thing will happen to you if you get shot. Again, if you lose your quarters by getting shot, your guards will stay there. Only do not appear new ones anymore.

## Similar games

Portal

Z

San Andreas

# Implementation

## Program architecture for real-time game

Programming real-time applications is discipline different to other types of software development. High emphasis is placed on 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.

Since the process has to be fast we need to do some calculations only approximately or asynchronously. Both of these techniques we will use in our game.

Now let’s see how to make a game architecture for our software. We adopt practices from XNA. It provides prepared process model for whole game. First we need to initialize our components, then load all needed content because loading can be very slow operation. Then main game loop comes. Main game loop is endless cycle between updating the game logic and drawing the scene. And at the end, as soon as the game logic decides the game is over, we end the main loop, unload loaded content and do whatever we want. For example exit application or restart it.



Picture Game life cycle diagram

Good idea is to have this process distirbuted into separate components. Because of clarity. We have several smaller modules running according the diagram metioned above: Town, Player, Opponent. Moreover the town component ditributes these operations into quarters and quarters into walkers, flying bullets, etc.

## Space and the game world

Before we begin model the bizzare world as it was defined, we can 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 vector space. Second debasement of the problem is that we consider only two dimensions. The third, height, will be added later only in selected functionalities. We do not need it everywhere. Finally the two-dimensional processing will be faster and that is what we want.

At the bottom of the space hierarchy we define basic geometrical elements: line segment, triangle and quadrangle (convex). Everything in our space will be based on quadrangle. Or more precisely, every object in the game has its projection into two-dimensional space as convex quadrangle. These quadrangles are used for collision detection.

Quite often we need to check if two objects are in collision. For example if a bullet hits a man. To get know it 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 – four subprocesses. Now the last thing needed 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 from the second triangle – nine subprocesses. We must not forget that for us one triangle inside the second is collision to.

Considering the game logic, the use of quadrangles isn’t 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 model (ex. robot) and we want to insert it into the game. Should we define all four corner points and scale model to fit corners into created quadrangle points? No. Better is to add next level into space hierarchy. We present game object. Game object is structure ready to use with 3D models and it is still simple enough to be in two-dimensional space. For work with varied 3D models 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 we are 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 space.

Now it is time to say how to represent our bizzare space. After all, what are our technical capabilities? We can display on the screen set of objects variously transformed by position in three-dimensional linear space, azimuth, scale and some projection parameters. So we need to use somehow classical 3D space. The idea is to split our bizzare space into parts which separately are vector spaces. Position of game object is now specification of concrete part of the world and coordinate vector from linear space of the particular part. Now is clear why we can on lower levels of abstraction use only classical linear space.

Back to the game objects once again. Game object provides basic info about everything in the game. It also provides projection into quadrangles: takes vector space coordinates from position, size and azimuth and calculates four corners. Now we can implement many of game object derivations: spatial objects carrying 3D model or flat ground objects and plates carrying only texture instead the 3D model.

We need to decide how to split our bizzare space into mentioned parts. We make it by definition of the game. We split it into individual quarters. Town quarter is subset of linear space and this division will show up helpful in further game logic implementations.

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

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

Could it be better? 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 subject. It is called space partitioning. There are used data structures like BSP Trees or Quadrant Trees in space partitioning. These techniques are based on search trees. In every node the space is divided into 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. Problems of this structure are moving objects. 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 thru the tree into the deepest leaf. But we are creating real-time game and we need these calculations fast. We like how quickly the space partitioning trees test for collisions 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 the collision test of all objects in the quarter takes about where is number of grid fields.

## Town generator

Before the game starts and after the content is loaded we need to create the world where our game will go on. Our game, because of its specific rules, has not any prebuilt maps. For every game instance we create the whole scene from scratch again.

First the input comes. And the only input is the number of quarters () that will be in our town. We prepare empty non-oriented graph with 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 what order do we have on them; it is just needed to have them joined. Also it is not needed to have them in cycle, so we don’t do a cycle – only simple path of length . 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 are 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 which interfaces (connection to nearby quarters) will be where. 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 inside formed by road and sidewalk. To the border road we connect interface roads. Inside the border road rectangle we have empty space. Using random generator we 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.

Since we have roads inside the quarter we can start putting in buildings and decoration objects. At first we built 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 interfaces. Next to one another – no spacing. Than we have empty rectangles inside the road network. These we fill by buildings with spaces between them. Or we don’t. However we want to.

Now we have done all the quarters and its interfaces 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 thru 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. 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. A\* algorithm commonly uses distance in Euclid metrics. But 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 we know is the distance between two interface path graph vertices – those from opposite quarters. This distance is constant. We have chosen it in 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 step into next vertex and heuristic there is always non-negative. To get to another quarter we must go thru an interface.

## Boxes, tools and action objects

Now we have 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. Tool will be a generic entity in our game that human can handle. But first we need boxes. Boxes will be first special objects in the game. Boxes react to collision otherwise 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 box for our game: toolbox and healbox. Healbox is simple, there is some kind of medicine and human who it takes turns into hundred percent healthy. Toolbox contains an instance of tool.

Tool is generally held by a human and it can do some kind of action. The only way we use tool abstraction are guns. 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” call.

Gun is derived from tool. Gun is instance of specific type. The gun type carries information about range, damage specification or standard bullet capacity. Gun, the instance, has information about load state and of course its type. Action of the gun is shot. The gun reads position of its holder and azimuth and puts bullet in the space. How to represent a bullet? First option is to simulate actual bullet object, small piece of metal flying thru the town, test its collisions and travelled distance and decide its fate. Problems of this approach mainly are 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. The performance problems are automatic weapons. 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 by one moment and the object can be shown for example for few milliseconds.

When we put one single object instead of bullet flight we will get set of objects the bullet goes thru. 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 not 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 we can test collision against other objects. We use the bisection method. We start with the gun’s range as length and take the set of colliders what we got from space partitioning collision system. Now we recursively 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 impulse. Active object is defined by description of an action it can do and distance from which it can be started. This 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.

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 during this action.

## People, reflexes and tasks

Object human has already been mentioned. We have prepared interactive content for humans. Now it is time to implement them. Human is much more complicated game object than those previously descripted. As it was written above, human is 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 info.

Now let’s go straight to the human behaviour. We need to make human acting his role in our game. The centrality of our effort we put into the every tick update logic. Every moment the human must decide what to do.

The really 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. For use 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. If the human see his enemy, the way is clear and he is in striking distance, he shots. If the way is not clear or the enemy is too far from the human, he makes a move toward to 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 of reflexes moves with the human, other reflexes or move-decisions are forbidden.

The next what comes after the reflexes is planned moving – tasks. Human has queue full of tasks what 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 abstraction for human to act his role in the game. One task has two basic features. It and only it determines whether it is finished 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 move task. It finds the nearest path graph vertices for the starting position and the destination and then searches for the shortest path through the town path graph between those two vertices. After these calculations it leads the holder through the waypoints until he reaches the destination. There 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. Infinity move tasks we give to quarter guards or walkers. The quarter will look more interesting, though there will be some movement. Next type of task is kill task. Every tick the holder is navigated toward to the target and if he is in striking distance, he shots. 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 then when the target dies. We implement also action object task. This type navigates the holder to specified action object and then makes him to perform the action object’s action. After this the task is complete. The last type of task is special. Temporary task is container for another task and 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 is returned result of request from the inner task.

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 we call update process of the selected task. It will move by the human and our behaviour stuff 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 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. That’s because the situation when the human is reaching his target and one of his enemies enters this quarter. The human does not have to leave his target, it would be dangerous for the human.

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.

## Opponent, task planning

The opponent is extension of human. We use the same logic about reflexes, task solving and post reflexes. What we put extra into opponent is task planning. Opponent plan his tasks to win the game and then he act like an ordinary human.

We add only two post reflexes to opponent. First is the actual task planning what is called if the opponent has empty task 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 quarter he is located in. Of course we add this action object task to the opponent only if he has not already task in this quarter.

The most interesting thing in opponent 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 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 that will be 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 info about quarter ownership – which quarters are owned by whom and 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 opponent’s quarters are good, owned by the player are bad. From the owned time we calculate number of guards in the quarter and multiply by it the quarter quality index. Health and damage are included in the calculation too.

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, kill the player operation and take box operation. Every operation calculates time that is need 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 fill opponent’s damage. 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 add.

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

Now we have everything for planning process. Forward planning is simple graph searching. We start at current game state and after considering all available operations we searches 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 we want to be fast with this operation. 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 and there will be free quarters, health and tool boxes to take and ability of killing the player, the searched tree has potentially nodes. It is definitely not possible to do this operation in regular update process of the opponent. But it is not necessary to have results from planning in the same update process as it was started in. Thus we run planner asynchronously. Task planning is separated operation, so it is not necessary to implement lot of synchronisation primitives. The only part that shares memory with the main thread of our game is saving plan into task list. Using task list is the only one part of the opponent implementation which needs synchronisation.

## Player, camera and game controls

Player-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 for game control and based on the caught ones call relevant behaving from human’s part of the player. Plus we calculate difference of mouse cursor position and rotate player object.

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 to do 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 to do it wrong. And the remaining quarters? In the quarter generating process we determined that the quarter border is filled by buildings. There is practically no view of other quarters except 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 fit together with our quarter.

## Settings, xml configurations and menus

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

Before starting the game we show to user windows with settings tabs and button for start playing. In video settings tab we let him choose screen resolution and set whether he want 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 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 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 play. Of course we must make our own versions of these files and add to game some default content because without it the game would be not able to run at all.

# User documentation

## Installation

## Start and settings

## Gameplay and controls

## Game ends

# Bibliography

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