1. Introduction

“101 Dalmartians” is a 3D action-shooter developed as the practical part of the lecture “Game Development” at the National University of Singapore (NUS) [link: 1]. The lecture took place from August to November 2005 and was conducted by Dr. Golam Ashraf. It covered the whole process of developing modern video games, from the basics of character and level design to the implementation. The emphasis was put on efficient and scalable algorithms for game physics and artificial intelligence.

The students were divided into groups of three and had to develop a game according to given specifications. The game concept, graphics, animations, implementation and documentation had to be created within the period from the 10th of August 2005 to the 9th of November 2005. My group, made up of Gilles Pierre Vincent Jaffier, Cedrik Pascal Sylvain Manzoni (both were exchange students from France) and myself, decided to take the Disney movie “101 Dalmatians” [Link: 2] as the base for our game.

This work explains common algorithms and methods used to create a modern 3D-game from scratch. “101 Dalmartians” will serve as an example and general experiences made while creating the game are described, but the emphasis is put on general algorithms that are used in a wide range of games. As most of the algorithms described in this work can be assigned to the domain of the game engine, game designers who are using a 3rd-party game engine will not have to implement these algorithms themselves. But an understanding of what the game engine does might come in handy.
This document will roughly follow the process of the creation of the game. First of all, the requirements to the game set by the lecturer will be explained, followed by the general concept and scenario of “101 Dalmartians”. After this, a short chapter will explain how the animations and graphics were created.
Game physics makes up the biggest part of this work. A chapter about space partitioning explains how to divide the world into logical areas, which is one of the most important parts to organize the game world in an efficient manner. The following five chapters explain how accurate, yet efficient collision detection can be achieved. This is not only necessary to let the characters move around in a realistic way, but also the foundation to create the perception of agents in the game. A chapter about heightmaps explains how to create the ground of the game level in a very easy way, and a chapter about camera control describes some collision-related special cases that have to be taken care of when creating a 3rd-person game.
After that, the importance of waymarkers and methods for path finding used by agents will be explained. Path finding could be seen as part of the artificial intelligence. However, most of the work in this project went into “outsourcing” computation from the agents into precomputed routes and heuristics that the agents can use, thus it will be treated in a separate chapter.
The following chapter describes and discusses techniques used for artificial intelligence, from a game designer's point of view. The emphasis lies on making agents appear to be intelligent, not on actually making them intelligent. Algorithms for modeling the agent's perception make up a big part of this, followed by the decision making process and steering, which refers to navigation on a microscopic level. A chapter discusses the need of a memory for agents. The final chapter within the Artificial Intelligence unit explains the concept of on- and off-screen behavior, to sacrifice a certain amount of realism in favor of speeding up the computation.
After that, the implementation of the actual game “101 Dalmartians” will be covered – including a description of the used rendering toolkit “Ogre 3D” [link 3] and a overview of the major classes used.
A discussion about the state of the project and the experiences made while creating the game will conclude this work.

2. Terminology

Some of the terms used in this work are ambiguous and used for several different concepts. To avoid misunderstandings, the most important terms will be clarified at this point:

- "Player" will always refer to the actual human that plays the game
- "Character" refers to the main character of the game being controlled by the player. This distinction between "player" and "character" is taken from "Rules of Play" [9].
- "Agent" refers to all autonomously moving objects in the game that are not controlled by the player. As many parts of this work are about the concrete game "101 Dalmartians", whose agents are all animals, the more common term "creatures" will be used from time to time.
- "Global world knowledge" refers to the concept that agents may receive knowledge about the world in other ways than their modeled perception. With global world knowledge, every agent always knows relevant information like where the main character and other agents are. Usually this decreases the realism of the game and therefore should be avoided.
- "Artificial Intelligence", in this work, refers to higher-level topics like decision making as well as to low-level mechanisms of perception and path finding.
- "Level" refers to a coherent "world" in which the games takes place at a given point of time. "Coherent" in this context means that all associated data is loaded when entering the level, all agents are "alive" and objects are only added or removed according to the game logic. In that sense, entering a new level does not necessarily mean that a "mission accomplished" screen will pop up or the game is interrupted in another way, but that a new set of objects or agents is initialized, and usually a new part of the world is accessible.

3. Game requirements

The following part is a short overview of the requirements the project had to comply with. The original 5-pages document can be found in the appendix.
The game had to be an 3D-game in an outdoor environment from the 3rd-person perspective. This means, the main character can be seen as part of the world, unlike in 1st-person-shooters (like Doom, Quake or Unreal), where the player perceives the world through the main character's eyes.
A growing civilization of creatures threatens to consume all food resources in the world and to kill the player. Three types of creatures exist within the civilization: the alpha female, the alpha male and the common creatures called “spawns”. Only the alpha female can give birth to new creatures, after mating with the alpha male. The goal of the game is therefore to kill the alpha female. The alpha male, on the other side, will protect the alpha female and provides her and her babies with food. Once the babies grow up, they leave the nest and will look for food for themselves and eventually face the player. Every creature has an own artificial intelligence, a perception and some internal variables like courage. They do not have global world knowledge. They are are individually weak against the player and thus prefer to attack in groups.

A part of the requirements on this game was that every group of students had to create all graphics and animations on their own (except for one group that had only two students and was allowed to use slightly modified characters of “101 Dalmartians”). The animation of the character and the agents had to use skeletal animation, created in the modeling software Maya.

The game had to be implemented using Visual C++. As the goal of the project was to study the core concepts involved in the creation of 3D-games, it was forbidden to use a game engine or libraries with a similar purpose, except for the rendering engine OGRE. Sound output in the game was optional, therefore using a sound engine and sounds and music from external sources was allowed.
The whole level had to be divided up into logical sectors for optimizing the efficiency of algorithms like collision detection, artificial intelligence, rendering and path finding. Collision detection had to be implemented between dynamic and static objects using hierarchical algorithms. The virtual camera had to avoid collisions with objects and had to move smoothly. The AI had to use hybrid rule-based and goal-based strategies and switch between different computation strategies for on-screen and off-screen agents.

4. Design

4.1. The story and theme of the game

The description of the creature's civilization – one mother and one father giving birth to a huge number of children and protecting each other from an enemy – reminded us of the Walt Disney cartoon “One Hundred and One Dalmatians”. In this classic, which was first published in 1961, an evil (human) woman named “Cruella De Vil” kidnaps many dalmatian puppies. Their parents, the dalmatian dogs Pongo and Perdita head out to rescue them [11]. We took this as an inspiration for the game setting, inverting the perspective: the player controls Cruella De Vil and has to kill the creatures. The game plays on a foreign planet, the creatures are extraterrestrial life forms. The title of the game integrates this idea as the name "Dalmartian" is a pun on the most famous aliens of all time: the Martians.

Fig. 4.1: The world

The game world is designed to look “comical”, with very bright colors and a peaceful-looking environment. It is an island with nice beaches on the one side and big mountains on the other. Healthy, green trees are growing on this island and rocks of different shapes are lying around. The world is populated by many cute, innocent-looking creatures, which look like a crossing between a dalmatian dog and the New Zealand bird kiwi.

Fig. 4.2: Fighting the creatures

In contrast to this idyll, the main character is aggressive and sadistic and has a very powerful gun as a weapon, which is nearly as big as herself. To emphasis her aggressiveness, she also has other ways to fight: she can fight with a machete, a short-range alternative to the gun. And out of the pure joy of being nasty, she can kick things and creatures around with her legs. Once in a while, she makes remarks like “so, will you move, or what?” to the player.

Fig. 4.3: Kicking creatures away

In the beginning, it was planned to include a background story to the game with cut-scenes triggered by certain events. The cut-scenes were supposed to be 2D-drawings in comic-style which tell a bit about the reason for her to kill the creatures. While the framework to display cut-scenes was implemented, the idea was discarded during the project as there was no time to create the drawings.

4.2. Creating the graphics and animations

The 3D objects of the game are created with the modeling program Maya [link 4]. Various export-PlugIns do exist for Maya, one of them [link 5] converts Maya models into a format readable by Ogre.
The creation of a model begins with some conceptional drawings on paper. Most important are two drawings of each object to be modeled – one image where you can see the object from the front, and one from the side. In Maya, such drawings can be imported, aligned to the axis and thus serve as a template for the following modeling process.
Given this drawing, the actual modeling of an object is straight-forward, however time-consuming. One of the basic geometric shapes provided by Maya is used as a base, and by splitting faces, extruding faces, deformations and other similar operations, the model approximates the shape of the drawing. In the case of humanoids and animals, usually only the left or right side of the creature is modeled manually, as the model can be mirrored after the completion of this one side.

The next step is to color the model. First of all, texture bitmaps have to be assigned to the faces of the model. Maya does this automatically to a certain extent. However in more complex models, pixels of the texture bitmap might end up being assigned to more than only one face. The effect would be that if one of the face is being colored, parts of the other face gets the same color. To prevent this, the assignment of the bitmap to the faces has to be adjusted manually.
Another design decision is the choice of the bitmap size. The bigger the size, the more detailed the texture can be, but the game will be slower and more memory will be used on the graphics card. Some approaches do exist to deal with the performance problem of bigger texture maps. For example, smaller versions of the bitmaps can be created, which can be used if the model is far away from the camera and therefore small on the screen. However, while this reduces the memory consumption on the graphics card, this comes with increased administrative efforts, even more memory usage in the main memory and if the whole system is not designed very carefully, strange graphic effects occur when the object gets closer to or more far away from the camera and the used bitmaps is switched.
The process of coloring can be done directly on the texture bitmap, or more conveniently in the 3D view using a 3D painter tool. It is some work, but again straight-forward.

Animating the model is the tricky part. The technique used is skeletal animation. Starting at a fixed point relative to the object's position, a hierarchy of logical bones and joints is defined within the mesh. Every bone stores its length and the relative rotation against the parent bone or the absolute rotation against the coordinate system if there is no parent bone. Therefore, if a bone in the upper part of the hierarchy is rotated (like the upper leg), the lower parts move along automatically.
Every face of the mesh is assigned to one or more bones, with weighted factors for each bone. Theoretically, every face can be assigned to a nearly infinite number of bones (and it is, from a logical point of view, with a weighting factor of zero for most bones). However most toolkits like Ogre do not support more than four actual assignments.
Again, Maya does an initial assignment. The bones can then be moved around freely to see if the skin is moving along correctly. For humanoid creatures, problems are likely to occur around the shoulders, as there are several bones in this area (two shoulders, the head, and at least one torso bone) and an automatic assignment hence is difficult. Maya provides a tool for adjusting the weighting factors of the bones to the faces.
For example, if a face that is located below the axle on the torso is moving “outside” of the body when the character is moving its hand upwards, the bone located at the elbow has probably a too high weighting and the bone inside the torso has a too low one. Once the influence of the torso bones is increased, the face will not move along with the hand anymore.
In the animation mode, Maya provides a time line for defining the animations by creating key frames in which the model has a certain posture (given by the position and rotation of the

bones). The frames inbetween are interpolated. An important issue when creating repeating animations (as for walking) is that there must not be any gap in the animation when it jumps from the last to the first frame of the animation. Therefore it is advisable to define a key frame at the end of the animation that is identical with the first one. Once the animation is completed, this frame has to be deleted, as the frame would be played twice otherwise.
For walking animations, another problem to be avoided is slipping on the ground. This happens while a character is standing on one foot which is moving “backward” (initially, the animation is done torso-centered). If this is not synchronized with the movement speed the character will have in the game, it will appear to slip on the ground. A Maya tool allows to emulate a movement of the model and therefore adjust the speed of the animation. If the character is supposed to walk or run in different speeds, either the animation has to be accelerated, or a new animation has to be created.

In the end, the mesh, the texture and the information about the animation has to be exported using the MayaExporter plugin by Ogre3D. To check if the export succeeded, the files can be opened using the OgreMeshViewer [link 6].

5. Game physics

After this short introduction into the creation of 3D-models, the more technical part of this work begins, explaining algorithms that are used within game engines. The main part of this chapter is how to create an efficient and scalable collision detection and how to deal with it in the game once a collision between the character, agents or the camera does happen.

5.1. Organization of the world / Space partitioning

One of the most important ways to improve the performance and scalability of a game is to divide the world into smaller areas. This partitioning can be used to to minimize the computation done by algorithms like collision detection or path finding. For example, to check if an object that is entirely contained in one area is colliding with any other object, it has to be checked only against objects that are (entirely or partly) in the same area. To do this efficiently, every area stores information about which objects are located within it. If the object moves, this information has to be updated.
To judge if a given partitioning is good or not in a given case, several factors are important:

- The areas should not be too big. The smaller the areas are, the less objects will be in it, reducing the cost of collision detection.
- However, the areas must not be too small and their shape and arrangement should correlate to the shape of the level. If the areas are too small, then a lot of updates about the location of the objects will have to be performed if the objects are moving around. It is also more likely for an object to be located in more than only one area at the same time, creating a computational overhead.
- If a lot of areas do not contain any objects at all, then resources are wasted.

The level can be partitioned on a 1-, 2-, or 3-dimensional basis. To partition on a 1-dimensional basis is usually not very helpful. The decision between 2- or 3-dimensional partitioning depends on the type of levels. Partitioning is most useful to efficiently prune away objects from computation that are not in the vicinity of a given object. So 3-dimensional partitioning makes sense if this can happen frequently along the height (one object is far above another one). This might be the case in complex space flight simulations, but as the degree of freedom is very limited along the y-axis in “101 Dalmartians”, a 2-dimensional partitioning is used here.

Another categorization of partitioning methods is to divide them into hierarchical and non-hierarchical methods. For non-hierarchical methods, every point in the world belongs to exactly one area. The underlying structures are therefore usually quite easy to understand and to implement. In hierarchical methods, the world is divided into a given number of areas. These areas can be, but do not have to be divided again into a number of sub-areas, and so on. This way, a “space partitioning tree” is created. These methods are more complicated, but sometimes better suited for big levels.

Three standard ways of partitioning will be introduced, with increasing complexity and increasing scalability. They are described for the 2-dimensional case, but the concepts behind them also can be used for 3-dimensional partitioning.
For illustration, a sample level (a l-shaped island with four objects on it) will be partitioned according to the given methods. To show the time complexity characteristics of each method, querying which area a given point belongs to will be taken as an example.
After these three standard ways, the approach used in “101 Dalmartians” will be explained.

5.1.1 Uniform grid partitioning

Uniform grid partitioning is the easiest one to implement. The whole world is divided into squares of equal size. Thus using squares of the side length of *l*, a level of size *w*h* is divided into *(w/l)*(h/l)* sectors. Querying which sector a point *(x, y)* is in is extremely easy and runs in O(1) time: *[Floor(x / l) | Floor(y / l)]*. The demerit is the memory usage, which has O(w*h) complexity, rising with the size of the level. If performs well if the objects are expected to be distributed homogeneous over the world. However if there are vast areas with no objects in it, while most objects are concentrated on certain areas, uniform grid partitioning becomes mostly useless (for a big *l*) or extremely memory consumptive (for a small *l*).

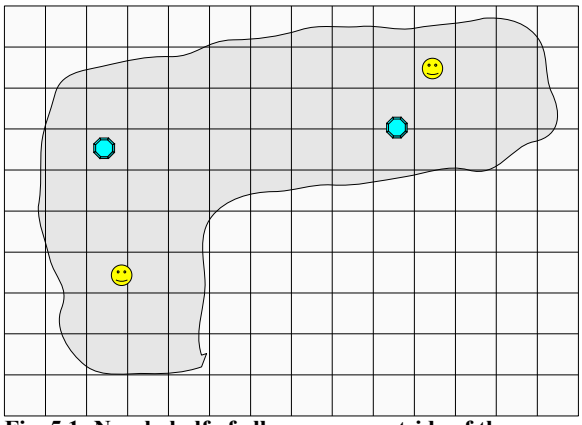

Fig. 5.1: Nearly half of all areas are outside of the island. They will never hold any information about objects and are therefore pure overhead.

5.1.2 Quadtrees

Quadtrees are a more flexible way to partition the world using a hierarchical structure. The world is partitioned iteratively: the whole world is divided into four rectangles of equal size. For every rectangle, the number of expected objects in it is calculated. If it surpasses a given threshold, the rectangle is divided up into four more rectangles, and so on. In that way, a tree is generated, with the final regions being the leaves. The big advantage of quadtrees is that those leaves do not have to be on the same level. So if there are wide areas with only little objects in it, it will not be divided any further, while areas with many objects in it are getting further divided.

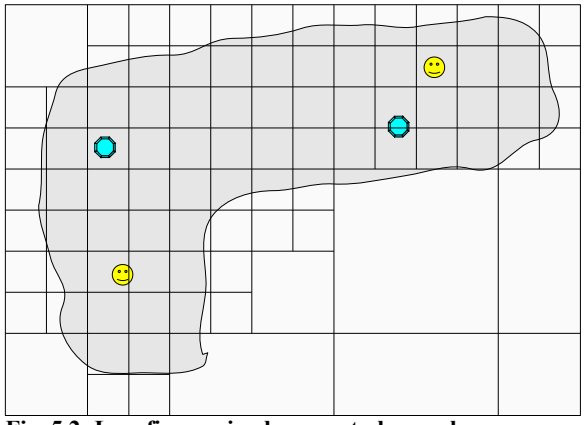

Fig. 5.2: Less fine-grained areas at places where no objects are to be expected.

Querying a point means walking through this tree starting from the root. On every node, the algorithm will check in which one of the four sub-areas the point is, and go on to this node until it reaches a leaf. With rising size of the level, querying a point therefore runs in O(log n). This is slower than uniform partitioning, however given the much smaller memory footprint, this additional computation is usually worth it once the level gets bigger.

5.1.3 BSP-trees

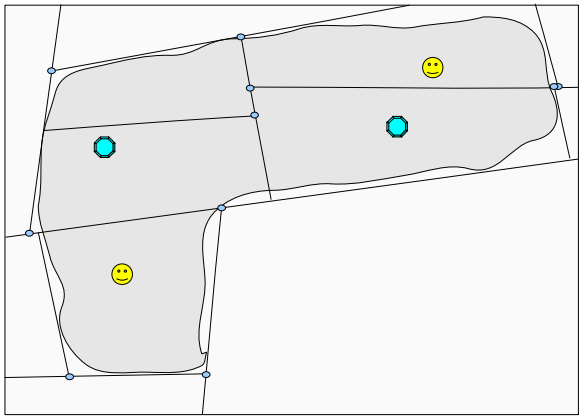

Fig. 5.3: With BSP-trees, the areas can resemble the shape of the level very closely.

BSP-trees are the most complex structure being introduced here. With BSP-trees, the level gets broken up into small pieces by recursively dividing the remaining area by a line (or a plane, in the 3D-case). The line can be defined freely, providing a line/plane-equation (*ax+by+cz-d = 0*). If the *(x/y/z)* is being substituted by the coordinate of a point, the side of the plane the point lies on can be decided by the signum of the result (if it is 0, it lies on the plane, the equation is satisfied). Constructing the BSP-tree can be done manually or automatically. If the BSP-tree will only be used for partitioning the level, it is advisable to construct it manually in a way to make it resemble the actual structure of the level. The flexibility of BSP-trees enables to define logical areas that approximate "physical" areas in the level. This can be a great help for path finding.
Querying which region a point is in works similar to Quadtrees – with the obvious differences that on every node there are only two sub-areas and the formula to decide between them is different. Time complexity is O(log n), too.

5.1.4 The system used in "101 Dalmartians"

"101 Dalmartians" uses a much more simple method of space partitioning: The level is mostly covered by nine manually defined rectangular areas. They are arranged as a linear list, so the time for querying a point is O(n). While O(n) is usually not a very desirable time complexity, it is no big problem for small levels with only few areas. With 9 areas, 4.5 checks have to be done on average to find the matching region. If nine areas were arranged in a balanced binary tree, 3.2 checks had to be performed on average (7 leaves on level 3, 2 leaves on level 4, therefore (*(7*3 + 2*4) / 9 = 3.2* checks). The difference was too little to justify the more difficult implementation of a hierarchical system. However, if the level was much bigger, with, for example, 100 regions, then the difference would be significant: around 50 checks in linear arrangement, and *(28*6 + 72*7)/100 = 6.7* checks in a binary arrangement on average.

5.2. Collision detection – the basics

The collision detection is one of the crucial parts in every 3D-game. Having accurate collision detection greatly improves the realism of the game and helps to create a good gameplay. If the player wants to go along a narrow passage that *looks* wide enough for the player, but he cannot enter it because a collision is detected although there should be none, this will be seen as a flaw in the game. It is even more irritating if the game does not detect a collision and the player ends up walking into a rock, or if his shots are missing the enemy although he aimed correctly.
On the other side, performing an accurate collision detection is extremely computation intensive and hence slows down the game. While algorithms to check for collisions between two complex meshes do exist, they belong to the world of movie production, not to the world of nowadays' computer games. So games usually have simplified models of the visible world used for

collision detection. Even with these simplifications and highly optimized algorithms, this part still makes up around 15% of the overall computation time in a modern 3D-game [1].

The basic technique of efficient collision detection is to represent the complex objects by a bounding volume or a combination of several bounding volumes that are, from a geometrical point of view, easier to deal with than the meshes. While the idea of meshes is basically the same – using vertexes, edges and faces as approximations to objects of our real world – their level of detail for rendering can be much higher thanks to powerful graphics cards.
In the simplest case, every object is represented by a single box that entirely contains the object, with the edges being parallel to the axis of the world coordinate system (Axis-aligned Bounding Box / AABB; see chapter 4.3.1.). With a representation like this, it is easy and efficient to prevent two objects from colliding. However, as in most cases the original object will not have the same shape as the box, there is a lot of empty space that is not part of the object, but will lead to a collision anyway. Two approaches are used to minimize this empty space, to make the bounding volumes fit the actual object more tightly:

- Using more flexible bounding volumes. Oriented Bounding Boxes (OBB), for example, can be rotated around all three axis.
- Using more than only one volume to compose a bigger entity.

Using these approaches, checking two meshes in the game for intersection can be reduced to a number of collision-checks between their bounding volumes. Taking two humanoids as an example, both might be made up of ten OBBs (two upper arms, lower arms, upper leg, lower leg, one torso and one head). To check them against each other, every OBB of the first object has to be checked against every OBB of the second one, which sums up to 100 checks in the worst case of non-intersection (if two OBBs are intersecting earlier, the rest of the checks can be skipped).
In the game, for every frame, every object has to be checked against a rather big number of different objects (against every other object in the world, in principle). With only ten such humanoids, 90 checks between the humanoids, thus 9,000 checks of the underlying OBBs have to be made. It is obvious that further optimizations are needed.

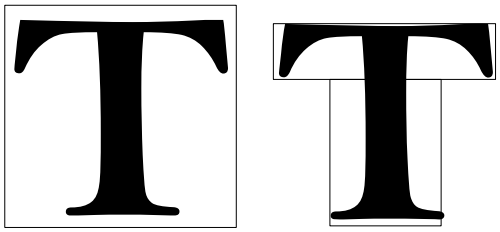

Fig. 5.4: 1st-level-OBB / 2nd-level-OBBs

To optimize this process, it is important to realize that most of the object will not collide. Therefore, it is more important to optimize the detection of non-intersection than the detection of intersection. Hierarchical systems are a very effective way to achieve this.
The first step to detect non-collision is to use the information given by a space partitioning system as described in chapter 5.1. If two objects are in two different regions, there is no need to do any OBB-checks between them as there can be no collision in the first place. It is important to note that a object can be in two or more regions at the same time, so in order to determine in which regions an object is, it is not enough to check the centroid of the object alone.
The second step is to represent the object itself as a hierarchy of bounding volumes. In the first level, the object is represented by a single big volume that does not fit the object very tightly. It should be rather too big than too small. The most important property is

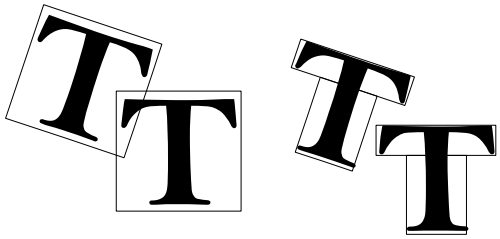

Fig. 5.5: The two 1st-level-OBBs intersect, but there is no intersection on the 2nd level.

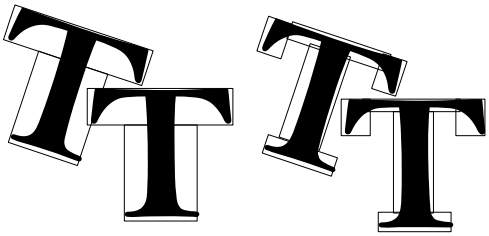

Fig. 5.6: An (incorrect) collision is detected on the 2nd level. To get the correct result, a 3rd level has to be introduced.

that every other bounding volume of this object is entirely contained in it. Given this property, if the two outer bounding volumes of two objects do not intersect, then there is no intersection between any volumes on lower levels or the meshes. If they do, then the bounding volumes of the next level are checked against each other. If no couple intersects, there is again no collision. If one couple does, then the algorithm recursively goes down the hierarchy until two leaves are colliding with each other.
In the worst case this procedure can be even more computation intensive than using no hierarchy at all, if all bounding volumes of the higher levels of the search tree are colliding with each other. In that case, all leaves' bounding volumes have to be checked against each other as in the linear case, plus the computation for the overlying bounding volumes. But as these cases are extremely rare, this is not a real disadvantage.

The big disadvantage of such a hierarchical system is the difficulty to design the bounding volumes. The most practical way to do so is probably to write a plug-in for Maya to support the creation of bounding volumes inside the modeling environment.
For animated objects like walking humanoids, one additional problem exists: their shape changes all the time (for example by moving the hands and feet), so the bounding volumes have to change as well. While it is impossible to specify the location of the bounding volumes for every animation state manually, the bounding volumes can be attached to the mesh's underlying skeleton. As the movement of surface of the hands and feed are defined by the bones they are assigned to, this should give a good approximation. The one thing to remember is that in a hierarchy, only the leaves are attached to the bones and the bounding volumes in the upper levels have to contain the leaves entirely. So after adjusting the position of the leaves to the bones, the whole bounding volume hierarchy might have to be readjusted.

5.3. Collision detection: Bounding Volume / Bounding Volume

This chapter explains algorithms for detecting intersection between different kind of 3D-objects. They make up the base for the collision detection as described in the previous chapter. Only algorithms to detect intersection between bounding volumes of the same kind are explained. In some games it might be effective to use different kinds of bounding volumes to approximate different 3D-objects. In that case, the number of involved algorithm increases quadratically with the number of bounding volume types used. As this would exceed the limits of this work, only homogeneous systems are considered.

5.3.1 Spheres

Spheres are the kind of bounding volumes that are easiest to handle. The bounding volume is described by a coordinate in the world (the centroid) and a radius. To check if two spheres are intersecting, the distance between the centroids c1 and c2 have to be compared to the sum of the two radii r1 and r2:

$$intersection = ((r1 + r2) > sqrt((c1.x – c2.x)^2 + (c1.y – c2.y)^2 + (c1.z – c2.z)^2))$$

(The C/C++ syntax is used.)

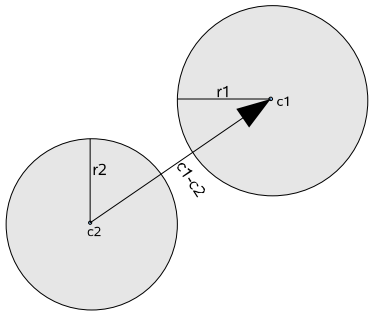

Fig. 5.7: (Non-)intersection between two spheres: |c1-c2| > (r1+r2)

As the square root is more complex to compute than the square, usually the squares are compared:

$$intersection = (((c1.x – c2.x)^2 + (c1.y – c2.y)^2 + (c1.z – c2.z)^2) < (r1 + r2)^2)$$

5.4. AABB

Axis-Aligned Bounding Boxes (AABB) are nearly as easy to handle as spheres. All six faces are parallel to one of the planes defined by two of the base vectors of the coordinate system. Every one of the 12 edges is parallel to one of the base vectors. To define an AABB, the coordinates of two opposing corners are sufficient. When deciding which two corners to take, it is advisable to take the one with the smallest coordinates (min_x, min_y, min_z) and the one with the biggest coordinate (max_x, max_y, max_z) – they are always opposing corners.
Given this information of two AABBs b1 and b2, checking for intersection is easy:

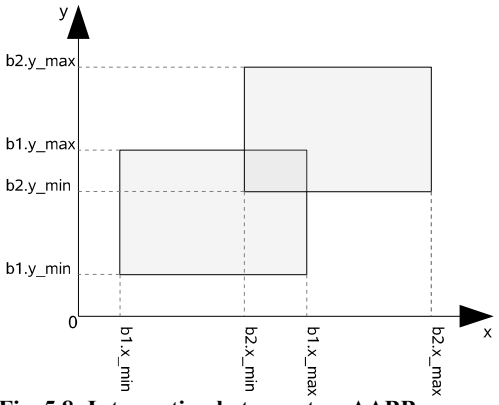

Fig. 5.8: Intersection between two AABBs.

```
intersection = (
        b1.max_x > b2.min_x && b1.min_x < b2.max_x &&
        b1.max_y > b2.min_y && b1.min_y < b2.max_y &&
        b1.max_z > b2.min_z && b1.min_z < b2.max_z
        )
```

(The comparison is supposed to return “false” as soon as the first comparison is false to save computation time. The formula is deduced from the Ogre3D source code.)

5.4.1 OBB

Oriented Bounding Boxes (OBB) are boxes as well, but more flexible as they can be rotated arbitrarily in the space and thus fit much closer to an object. An OBB can be defined by its center and three perpendicular vectors pointing to the center of three adjacent faces. Another way is to provide a center, the length of the three sides and a rotation quaternion (or any other structure suitable to define a rotation).
Checking if a single point is inside or outside of the OBB is simple: given the OBB defined by the centroid c and the three perpendicular vectors v1, v2, v3, the point p is inside, if:

```
inside = ( |(c-p)*v1| < 1 && |(c-p)*v2| < 1 && |(c-p)*v3| < 1)
```

* is the dot product

Checking if two OBBs are intersecting is more complicated and several approaches do exist. A concept easy to understand is to check if any of the edges of one OBB is intersecting with a face or edge of the other OBBs. If there is an intersection, the two OBBs do intersect, if there is none, then only one more check has to be performed for the case that one box is entirely contained in the other box. Checking if the centroid of a box is inside the other box provides the necessary information. While this method is conceptually easy to understand, a lot of single checks have to be performed, checks of different types (edge/edge, edge/face, point/OBB).

The Separating Axis Theorem (SAT) provides a more "homogeneous" algorithm to check for intersection using less computation and can be extended to any convex polyhedra.
For OBBs, the SAT defines 15 axes. The boxes are "projected" on them. If the projections of the OBBs do not overlap on any one of these axes, then there is no intersection. If the projections overlap on every single axis, then the OBBs do intersect.
The 15 axis are defined as follows:

- 3 axis that are orthogonal to the 6 faces of the first OBB, and the same for the second one.
- For each pair of base vectors of the two OBBs (3 vectors for each OBB, thus 9 combinations), the vector orthogonal to both vectors is an axis.

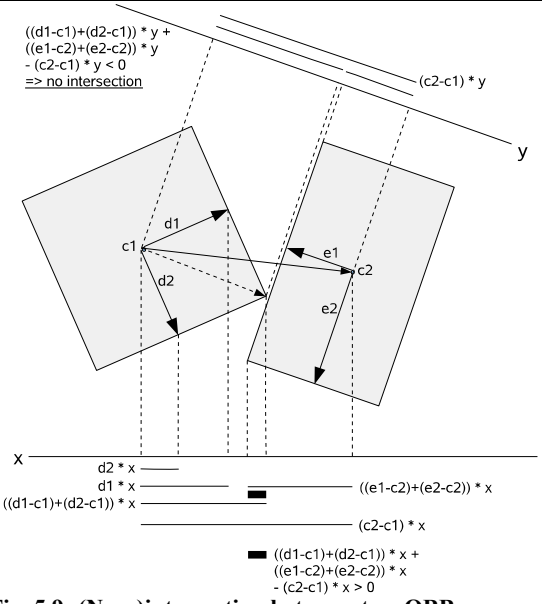

Fig. 5.9: (Non-)intersection between two OBBs.

Note that the axis do not have an actual location in the space, they are only directions defined by a vector. The vectors of the axis do not have to be normalized, as only the dot products will be compared.
For each axis a, the following steps are (theoretically) done:

- The vector between the centroids *(c1, c2)* of the OBBs is projected on the axis. The projected distance d between them is "*d = (c2 – c1) * a*".
- The vectors from the centroid of the OBB to its 8 vertexes *v* are projected on the axis; for each OBB this will give 8 values "*dv = (v – c) * a*".
- The vertex that results in the most positive value is taken for each OBB. The most positive and the most negative value of an OBB are always the same except for the signum.
- If "*d > (dv1 + dv2)*", then there is no overlap, thus no intersection of the OBBs. If "*d < (dv1 + dv2)*", then there is an overlap for this axis. This does not automatically mean there is a intersection, so the same steps have to be done for the next axis.

Like this, 17 dot products have to be calculated (distance between centroids, and for 8 vertexes of each OBB). To proof an intersection, this has to be done 15 times, which results in 255 dot products.
This can be reduced, using the special properties of a rectangular box. The vectors from the centroid to the centers of 3 faces are given. Using them and their negative counterparts (which are the vectors to the other 3 faces), the vectors to the 8 vertexes can be calculated by summing them up, using the 2^3=8 combinations of positive/negative vectors. The dot product of one vertex vector and the axis (*dv*) is the same as the sum of the dot products of the three base vectors *(b1, b2, b3)* used to create the vertex vector and the axis. Thus the most positive value of dv can be calculated by:

$$dv = |b1*a| + |b2*a| + |b3*a|$$

This reduces the number of dot products needed to get the most positive value from 8 to 3, and therefore the total number of dot products to be calculated to proof a intersection to (1+2*3)*15=105 (150 less than before).
In most cases, there will be no intersection. For that case, 8 axes have to be checked on average until one is found with no intersection.

5.5. Collision detection: Bounding Volume / Ray

A ray is defined by a origin *ro* and a vector *rv*. Rays are commonly used to check if an agent can see an object. In that case, the origin is the position of the agent and the vector is pointing to the object. Chapter 7.1 describes the algorithms for visibility detection in detail.

5.5.1 Spheres / Rays

To check if a ray intersects a sphere, first of all it has to be checked if the origin of the vector is within the sphere. If it is, then there has to be an intersection.
If it is not, then the dot product *d* of the vector from the ray's origin to the centroid *sc* of the sphere, projected onto the ray's direction vector has to be calculated:

$$dp = rv * (sc - ro)$$

If it is negative, then the sphere lies on the "invisible side" of the ray, so there can be no intersection. If it is positive, then the closest point on the ray to the centroid can be calculated by $p = dp*rv$. All that is left to do is to check if this point is within the sphere. With the sphere's radius being *sr*:

$$x = ((sc.x - p.x)^2 + (sc.y - p.y)^2 + (sc.z - p.z)^2)$$
$$intersection = (x < r^2) \quad [7]$$

5.5.2 OBB / Rays

Only an algorithm for intersection between OBBs and rays will be explained, as this works for AABB/ray-intersection as well. For AABBs, a more efficient intersection test using Plücker Coordinates is described by Mahovsky and Wyvil [8].
OBB/ray-intersection can be done using the Separating Axis Theorem [6]. Both the OBB and the ray (infinite line segment) are projected onto the axis. Six axis have to be checked:

- The three base vectors of the OBB.
- The three cross products between the base vectors of the OBB and the direction vector of the ray.

However, as rays are infinite into one direction, they cannot be seen as a "degenerate OBB" [Link: 17] like normal line segments. As there can be no centroid for the ray, the overlap test has to be modified slightly. In this case, the axis needs an exact location in the space, defined by the given vector *v* and an arbitrary point *p*. The overlap test can be done like this:

- The origin of the ray is projected onto the axis ($aro = p + v * (v * (ro - p))$).
- A different arbitrary point on the ray is projected onto the axis (e.g. $arv = p + v * (v * (ro + rv - p))$).
- If *aro* == *arv* then the ray is orthogonal to the axis and only this point is the projection. Otherwise, *aro* splits the axis into two halves – the one with *arv* in it is fully covered by the projection.
- In the same way as the ray's origin, the eight vertexes of the OBB are projected onto the axis. If any of them lies on the same half of the axis as *arv*, then an overlap exists.

If an overlap exists for all six axes, then the ray intersects the OBB. If a non-overlap is found on any axis, there is no intersection and the algorithm can stop.

5.6. The system used in 101 Dalmartians

101 Dalmartians uses a 3-step collision-detection, including the information provided by the partitioning of the level, AABBs and a stripped-down version of OBBs.
After an object is moved internally, it is checked for collision against every other object in the world. If there is no collision, the movement will be kept – if there is, then the movement is canceled. The three steps are:

- First of all, the information of the level partitioning is used to ignore objects that are not in the same region as the reference object. As there are 11 regions, this reduced the number of checks by 91% (if the objects are homogeneously distributed over the regions).
- In the next step, the AABBs of the objects are checked against each other. Ogre3D generates AABBs for every object in the world automatically. The documentation of Ogre states that this data is not supposed to be used this way, as the data is only an internal cache that is usually only updated while rendering and does not include the changes after the last rendered frame. However, there is an (also internal) function that forces an update, so after calling it manually the AABB indeed does include the movements from the current frame. As the AABB is guaranteed to include the whole object, there can be no collision between two objects if their AABBs do not collide. So in most of the cases (the objects are not very close to each other), no OBB-check has to be performed.
- As already stated, 101 Dalmartians uses a simplified version of OBBs for close-range collision-detection. In that, the three lengths of the box can be defined and the box can be rotated around the y-axis, however not around the x- and z-axis. This makes them less flexible, but using the Separating Axis Theorem, less axes have to be checked to show (non-)intersection, which makes the algorithm faster. The main reason for this decision however was another one: there was no time to implement a Maya-plug-in (as described in chapter 5.2) or an algorithm that creates OBBs automatically from the faces and vertexes of the object. So the OBBs had to be defined manually using a self-written OBB designer. It was predictable that this manual part was about to bring in some major inaccuracy. In addition to that, the OBBs are static per object and do not take into account the position of the bones of the object. With this kind of inaccuracy, implementing three degrees of freedom in the rotation appeared to be an overkill. As the characters itself only rotate around the y-axis (when turning left or right), this was the one rotation implemented. In the end, every object is composed by 1 (e.g. the main character) up to 11 (e.g. the nest of the alpha-female) OBBs, not using any further hierarchy. So to check the character against the nest takes 1*11=11 OBB/OBB-checks (given that there is a intersection on the AABB-level).

The reached accuracy was sufficient in most cases, given the time constraints of this project. It it possible to walk under the trees (the OBB describing the tree trunk is smaller than the OBB describing the treetop) and between the pillars of the arch (which was designed to show the advantage of the OBB-approach used over simple AABBs). The one object with really bad accuracy is the nest, as its round shape is very complex and it was designed very late in the period of deadline-panic.

Even with all these simplifications used (less degree of freedoms for the OBBs, manually adjusted, using no OBB-hierarchy and not taking into account the internal movement of the characters like raising hand), the system still gives a satisfactory result in this case, because of the limited freedom of the game itself. If the character could move around more freely, like flying or jumping high, or the camera could be moved around by the player, the flaws became more apparent: the player is stopping a bit *before* she reaches the tree trunk, and the round shape of the tree top is not reflected by the OBBs very well.

5.7. Collision Management

One more detail concerning gameplay and collision detection was tricky: the rotation of the player when moving left or right with the mouse. It is possible that a collision occurs after this rotation. The easiest way to handle this case is to abort the rotation and do nothing. While this prevents the collision, it has some negative impact on the gameplay, as it is disturbing for the player not to be able to move the way he wants for no apparent reason. So in that case, the rotation is done anyway, but with an automatic change of the character's position: Small moves to the front, left, right and back are done (taking the slope of the ground into account) and checked for collision. If one of this movement results in a state without collision, it is done. In the very unlikely case that all four movements result in another collision, the rotation is aborted and it is up to the player to get out of this mess. While playing the game for many hours, this occurred only in two kind of situations: when the player walks under a tree while the ground has a slope going up on the side of the tree the character is on (the automatic movement tries to move the character away from the trunk, however due to the slope the head of the players bumps against the treetop), and when the character is surrounded by creatures. In the latter case, she can free herself by kicking the blocking creatures away.

A related gameplay issue is the “sliding along a wall”: when the character runs against a wall, but not frontally but only slightly while walking rather along the wall than against it (the angle between the character's movement and the wall is smaller than 45°), the character is not supposed to stop, but to slide along it. This is done by the same technique as described above: if the collision occurs, the game checks if a movement to the side (movements to the front and back are not helpful in this case) can resolve the state of collision. If yes, the character is moved. In that case, it is important to keep in mind the issue of the maximal walking speed of the character to prevent some kind of “Doom running trick” (in the old 3D-shooter DOOM it is possible to run faster by walking diagonal): after this automatic movement, the vector between the original position and the position after the movement has to be calculated, scaled down (in the method described above, the automatic movement is always orthogonal to the original movement, so the resulting movement is bigger than the planned one) to walking speed, and checked against for collision.

5.8. Heightmaps

Heightmaps are a method to create the ground the character and the agents can walk on. The principle is simple: for every (x/z)-coordinate, the ground has exactly one y-coordinate. Therefore, the relevant information about the ground can be stored in a two-dimensional array, a heightmap. By this, creating the ground becomes very easy: a gray-scale-image can be used, with the x/y-coordinates of the image representing the x/z-coordinates of the 3D-world, and the gray-values representing the height. To edit this heightmaps, any paint-program like Gimp or Photoshop can be used. Mountains in the world are represented by areas of light gray or white, slopes by color gradients.
A second image of the same size is necessary to define the color of the ground at a given point. Ogre3D supports heightmaps when using the TerrainSceneManager (which is part of the “OctreeSceneManager”-PlugIn delivered with the standard-package of Ogre). Every pixel in the given heightmaps corresponds to one point in the level and its height. The height at a given x/z-coordinate can be queried using the “getHeightAt(x,z)”-function, for example when a character moves to a new position. If the height of a pixel is queried that does not have a corresponding point on the heightmap, the height is bi-linearly interpolated by the surrounding points.

Fig. 5.10: Heightmap of 101 Dalmartian's level.

Fig. 5.11: Texture Map of 101 Dalmartian's level.

Using heightmaps greatly eases the creation of the ground of the level, but comes along with heavy restrictions on the level design: complex structures like the famous cave behind the waterfall, or even bridges are impossible to create using the plain heightmap-approach.

In the level of 101 Dalmartians, bridges (or rather wooden boards) were needed in order to have a way over the river that separated the left and the right part of the level. This cannot be done using heightmaps, because you have to be able to look at the river *under* the bridge. So a new layer was put over the Ogre-getHeightAt()-function to separate the height information used for rendering the ground and the information used for obtaining the height for the characters. In most cases, the information is identical, but in the case of the bridge, the height for the characters (given by the bridge) is above the height of the ground (given by the river).
The big limitation still remains: although it is possible now to *look* under the bridge, it is still not possible to *walk* under it, as there cannot be two accessible places for one x/z-coordinate. There might be tricks to by-pass this limitation on a small scale as for the bridge, however they usually end up to be very inelegant. (For example: providing the wrapper around the Ogre-getHeightAt()-function with the current y position of the character. The function can return either the bridge- or the ground-height, whichever is closer to the given height.)

There is one more gameplay-related design-decision concerning heightmaps: what happens if the player walks towards a steep cliff? He can either be stopped, or fall off the cliff (probably involving health damage or death). In 101 Dalmartians, it is not possible to jump down a cliff or a bridge or walk along any path that is too steep.

The steepness of a slope can easily be calculated: get the difference of height between the current and the desired position, and divide it by the distance between the points (not taking the height into account). If the value surpasses a given threshold, the slope is too steep. However, this is not enough, as this only gets the slop along the axis the character is moving. If there is a very steep cliff (steepness factor "*s"*), the player could still walk it up, by taking a very small angle "*a"* to the cliff instead of trying to walk up frontally. The calculated steepness is "*s * sin(a) = 0*", if the player walks along the cliff (*a=0/180*), and it is equal to *s* if the player tries to walk it up frontally (*a=90*). For a small *a*, the factor could be below the steepness threshold and make the program assume the character can walk on – which is just unrealistic. A simple work-around is to calculate the two steepnesses orthogonal to the walking direction, and compare the biggest result to the threshold. If "*s * sin(45°)*" is bigger than the threshold, it will not be possible to walk it up anymore.

5.9. Camera control

In 1st-person games, camera control is easy: the camera is fixed at the position of the character's eyes. The character is invisible, and a collision between the camera and game elements cannot happen.
For 3rd-person games, camera control is a bit more complicated, as the camera and the character have to be treated independently to a certain degree. While the player controls the character, the game has to come up with a way to move the camera smoothly in a way that the player knows where the character is and sees the most relevant part of the character's environment.
In the simplest case, the camera is at a fixed position behind the character, in most cases slightly above the character's head to get a good view on what lies in front of the player. However, a collision test for the camera is needed: if the character stands in front of a wall, with her back to the wall, then the camera would vanish within it. Several approaches can be used to handle this case – the camera could move closer to the character, or even changing to a 1st-person perspective temporarily. Alternatively, it could move upwards on an elliptic course around the character. The most important thing is not to make this switch too sudden, which would irritate the player. Rather, the camera should gradually move to the desired place, accepting the drawback that the vision is occluded for a moment.
In “101 Dalmartians”, this way of camera control was used. The fixed position relative to the character's position made it possible to use the camera for aiming and shooting. It was a game design decision to let the human aim himself and not to use an automatic aiming system as several other 3rd-person games like “Tomb Raider” do. A cross-hair is in the middle of the screen, and the player will shoot at whatever the cross-hair is targeting at. This was fairly easy to implement, but makes the game a bit hectic: the camera moves jerkily around, as jerkily as the player's movements are.

Some games prefer a smoother camera control. If the character stands still first, but then starts to run, then the camera should not move as quick as the character immediately, but accelerate slowly to give smooth camera movements. In that sense, the camera gets undocked from the player. To achieve this, the camera's current position and orientation, the goal position and orientation (which is identical to the camera's position in the fixed position's case), and the movement speed and direction position- and orientation-wise have to be kept track of. If the difference between the current and the goal parameters are too big, the movement speed has to be increased (gradually), if it is little, the movement should be slower.
A common way to achieve this are so-called PID-controllers, “proportional integral derivative controllers”. PID-controllers are basically a formula that returns the new position of the camera based on its current position, the goal position and its current velocity. As the way a camera or any other object controlled by a PID-controller behaves in terms of speed and stability only depends on three factors provided by the game designer (one for the proportional, the integral and the derivative part of the formula – hence the name), it is comparatively easy to get good results quickly. While PID-controllers will not be covered in detail here, a lot of articles and examples about it can be found, for example in [Link: 20].

6. Path planning

6.1. Waymarkers

Waymarkers are an essential part of designing the behavior of agents in a 3D-world. While it is possible – and necessary, to a certain point – to give the agents perception of the world and the ability to draw conclusions from this percepts (the most important one is probably the ability to see the main character when she approaches), it is an extremely complex task to design an "ideal" artificial intelligence that acts reliable using only individual percepts, memories and experience. Even if it could be built, the computation involved would be too time-consumptive to simulate many agents in a real-time game.
For example, just assume a creature is standing at one riverside and wants to go to the other side. Going straight through the water obviously does not work, so it has to find the bridge that happens to be 100m away, leading over the river. The creature has to find out: is there a way to the bridge and from the bridge to the destination? Is the bridge actually traversable? And how to avoid obstacles that are in the way between the current position and the bridge, or the bridge and the destination?
Waymarkers are a "walkaround" to such problems. Basically they are a set of arbitrary data assigned to a location in the level. They provide abstract meta-data about the surroundings to the agents and by that release them from the duty of drawing conclusions based on pure perception. In a way, a tight net of waymarkers can be seen as a "meta-level" of the actual level.
Path planning problems like the one mentioned above are a common use for waymarkers. In the river-example, there could be a waymarker *w1* close to the original position of the agents and a waymarker *w2* close to the destination. *W1* stores information about how to get to *w2* – so the creature immediately knows there is a bridge close-by and can choose this path.
However, waymarkers are not limited to path planning. Examples for further uses are:

- Information about properties of the environment. A waymarker within a cave might indicate that this is a good place to hide, but, once the agent is seen by the player, a bad place to run away to (because it might end up being trapped there).
- Instructions for the agents. On the river at w1, the instruction could be "*IF (destination = w2 && has_ability(agent, "super_high_jump")) THEN agent.super_high_jump(w2)*". For example, as Perdita cannot move in "101 Dalmartians", Pongo has to drop off the food for her at a specific point. He walks to a specific waymarker, and then gets the instruction to rotate to 270° in the world coordinate system and then drop the food.

Using waymarkers extensively is a good way to save performance and ease the programming. However, when used too much, they come with a drawback: the agents might end up knowing too much about their environment, which does the opposite of the original intention to create a realistic game. Creating intelligent and interesting agents is not only about giving them as much information as possible, but also about giving them non-awareness about certain things and including some randomness and errors to their behavior. A rule of thumb is to include only this kind of information in a waymarker that could be easily recognized by a real living creature / human in the same situation. If the bridge over the river is only 100m away and visible from this point, can be included to the provided information. However, if the only bridge is 2km away and cannot be seen from there, it might make a game more realistic to make the agent wander around clueless into the wrong direction (unless it has individual memories about the level, of course).

6.2. Waymarker-based path planning

At least two sets of data are necessary to do successful path planing based on waymarkers: the coordinates of the waymarkers in the level, and the traversable connections between them. A traversable connections exists if there is no obstacle in the straight way between two waymarkers, or the obstacles can easily be avoided by the agent using some local steering mechanism.
If there is no separated area in the level that cannot be reached at all, then, using these connections, every waymarker should be reachable from any other waymarker over a number of in-between waymarkers. A lot of well-known algorithms exist to find the shortest path between two waymarkers – including plain breadth-first, A*-search and Dijkstra's algorithm.
However, two major design decisions have to be made before beginning to implement the search:

- Should the path be calculated on-the-fly when it is needed, or should it be precomputed and stored as a lookup-table? The on-the-fly-solution may result in a lot of computation to be done during the game if there are a lot of waymarkers. The lookup-table, on the other side, has two big disadvantages: first, as the shortest path from every waymarker to every other one has to be stored, the memory consumption increases quadratically with the number of waymarkers, which can lead to a huge memory footprint. Secondly, they are less flexible. To illustrate the advantage of a flexible system, imagine a very narrow path between two waymarkers. It might be big enough for a small critter, but the big fat monster would get stuck in it. So the connection between those two points exists for the small, but not for the big one, which results in different shortest paths. This fact cannot be represented by using one lookup-table, so either there has to be a lookup-table for every type of agent, or the small paths cannot be used at all.
- Will the search-algorithm work on all waymarkers, or will a hierarchical approach be used to reduce the computation time? Using the hierarchical approach obviously results in a more complex system (and more coding), but unless there is really a very small number of waymarkers, this is usually worth the effort (both the computation time and the memory usage will be reduced).

The same structure to divide the world into several logical areas for collision detection can be used for path finding. In the case of “101 Dalmartians”, around 350 waymarkers are scattered over 9 areas. So, around 40 waymarkers are in each area.
Now the path finding problem consists of two parts: to find a path within one region, and to find a path from one region to another one. The Intra-region-search is exactly the same as without any partitioning, just with a reduced number of waymarkers.

In “101 Dalmartians”, a slightly modified version of A*-search is used to precompute the Intra-region-routes into a lookup-table. The route from every waymarker in a region to every other waymarker is computed individually, using a search tree. The nodes of the search tree are the paths to waymarkers. The cost function for the search is the distance between the two waymarkers in the level:

$$f(w1, w2) = sqrt((w1.x-w2.x)^2 + (w1.y-w2.y)^2 +(w1.z-w2.z)^2)$$

The fringe of the A*-search is implemented using the MultiMap-structure of the C++-STL – the nodes of the search tree are the values and the assigned costs the keys.
The algorithm starts with the first waymarker, an empty path and a cost value of 0. In each step, the the node with the lowest cost value is taken. If the last waymarker of the path is the goal waymarker, then the algorithm is finished. If it is not, then the possible follow-up paths, the child nodes, are created and the cost values assigned. If one of the newly created paths is leading to an already visited waymarker, then it is immediately discarded, to prevent loops. Before adding the children to the fringe, one more check is performed: if there is already a path

stored in the fringe that leads to the same waymarker as one of the children, then either this path is deleted from the fringe or the child is discarded – whichever has the higher cost value. The remaining children are added to the fringe and the algorithm starts over again.
That way, all possible ways from one waymarker to another one are computed and stored in a cache file. However, to save space and memory, not the whole path is saved, but only the first intermediate step - if the shortest path from w1 to w7 goes over w2 and w5, then only “From: w1 To: w7 Next: w2” is saved. This is enough: a separate entry for the path from w2 to w7 exists and will be “From: w2 To: w7 Next: w5”, and one entry “From: w5 To: w7 Next: w7” - so the path can easily be reconstructed using only the next step for each waymarker.

The Inter-Region-region part is a bit more tricky and includes two big design decision: is it enough to have only one point where the agents can go from one region to another specific one (called “boundary point”), or are there more points? The boundary points can either belong to both regions (thus breaking up the 1:n-relation between region-waymarkers and changing it to a [1..2]:n-relation), or there are two different waymarkers at the same location, belonging to different regions (which comes along with some messy implications when trying to figure out the closest waymarker to a given point, as there are suddenly two of them).
If there is only boundary point between two regions, then going from a point at one region to another point at an adjacent region is straight-forward: look up the path from the origin to the boundary point within the first region and walk to it. Then look up the path from this boundary point to the destination within the second region and go on.
If there are multiple boundary points for one border, then the agent has to decide which one to take by summing up the distances to walk (path costs), compare them to each other and choose the shortest path.
The second decision involves the case that an agent might want to go to a region that is not adjacent to its current one and there might be different ways of reaching it, using a different set of boundary points. Two ways of handling the path costs do exist – one is more efficient, the other one is more accurate. The efficient way is to make the decision which regions to trespass independently of the actual path cost. The cost of every border-crossing is 1, thus the agent will try to minimize the number of border-crossings. The paths between the regions can be stored just like the paths between the waymarkers: “FromRegion: 1 ToRegion: 2 NextRegion: 2”.
In most of the cases, this will indeed result in the shortest path. However sometimes there might be cases where there is a shorter way than going to the boundary point of 1/2 first, that goes over another region (from another pair of waymarkers in region 1 and 3, going to the boundary point 1/2 will still be the shortest way). To get the shortest way for sure, a search has to be performed with the nodes of the search tree being the paths over different sets of boundary points. In this case, however, there is no practical way to precompute all possible combinations, thus the search has to be performed during the game.

“101 Dalmartians” uses the simple way: only one boundary point and precomputed inter-region-lookup-tables (that do not always give the real shortest path). The level was designed with this constraint in mind: the logical regions correspond to “physical” regions in the level, and the one boundary point between two regions is indeed the only physical way to get from one area to the other one. For example, the hideout of the alpha-female on top of the mountain (refer to fig. 5.11) is one region, which is only connected to one other region by the tiny mountain path which leads to the shore of the western island. From the island, there are only two ways to the mainland, using the two bridges leading to two different areas on the mainland. So if an agent wants to go from the nest to somewhere in the mainland, it has only few choices anyway, so the algorithm hardly can choose a longer way than actually needed. And even if it did, it wouldn't matter in this game: traveling from one region to another usually takes them a long period of time, so before they arrive they will get hungry anyway, discard their original plans and look for food instead...

7. Artificial Intelligence

7.1. Perception

When creating the artificial intelligence of the agents, giving them a realistic perception is one of the most important parts in order to create interesting gameplay. The reason for this is that in those parts of the game where the main character is hiding from the agents or sneaking up to them, the player will most likely notice flaws in the agent's intelligence. If the agent is far away, minor mistakes will not be noticed, and in active combat the player will be too busy to notice. However, if the character shoots from right behind and the agent does not hear it, it will immediately appear to be very stupid. Or if the character is hiding behind a big rock and the approaching agent can see him anyway, this will be rather disturbing. Of course, both kinds of behavior can occur intentionally - for example if the agent is deaf or, for the second example, has super-natural powers, x-ray-eyes or something like that. But given the success of games whose popularity rely nearly exclusively on the sneaking-part (e.g. Metal Gear Solid), it is usually a good idea to give the player a way to take a clever, sneaky way to approach the agents in order to take care of them.

Agents can have different kind of senses. The most important ones are the sense of seeing and hearing, followed by the sense of smell (when an agent can smell if the character has been at the same place before). The sense of touch and taste are less likely to play a big role in the gameplay (although the good taste of the character's flesh might be a big motivation for some monsters).
All perceptions will usually deal with one question: does the agent notice the presence of the character? To model the awareness of other agents based on perceptions might increase the realism of the group behavior (if an agent does not know about the presence of another agent, it might decide to avoid the character due to its own individual weakness, although there actually is another agent nearby). However, these situations do not occur frequently and most likely the player will not notice the minor inconsistency if the agent are given global knowledge about each other's position. To make sure this is not seen as an inconsistency, the agent might make some noise when noticing the character, so it is implicitly explained why the agents know about each other (they can hear each other's noises).

The sense of hearing is the easiest sense to implement, as it can be modeled less accurately, especially in out-door environments. The hearing is usually modeled as omni-directional and not influenced by objects between the source of the sound and the perceptor. Every object in the world can be a sound source, constantly emitting sounds of different loudness (zero loudness, if no sound is emitted at all). The loudness of the sound decreases quadratically with the distance from the source, and if it surpasses a given threshold, then the agent will notice it. Thus:
$notice = ((loudness / distance^2) > threshold)$

The threshold depends on the hearing capabilities of the agent, but it can also depend on the environment - if dozens of other noisy creatures are around and it is raining heavily, then the threshold is higher. An elegant way to take this into account is to sum up the loudness of all sound emitters around and check how much of this overall sound is due to the source to be checked.
$notice\_x = ((loudness.x / E\ i=0...n(loudness.i)) > threshold)$

The loudness of the environment (for example a loud waterfall whose noises might drown other noises) can be modeled either by "invisible sound sources", or be provided by nearby waymarkers (interpolating the loudness given the information of the 1...n closest waymarkers). For hearing, it might be worth to include a certain uncertainty in the information about the position of the sound source. Humans will get the approximate direction and distance of the sound source, but by far not as accurate as by seeing the source with the eyes.

The sense of smell is not used very often in games, as it gives the agents only little information – basically only the information that the character has been there not too long ago. The agents might get alerted if they perceive the character's scent and go on patrol. The only tricky part about the smell is how to store the information about the character's scent.
If the perception of the agents will only check if the character can be smelled, then the most convenient way is to store it in the character-object. The trace can be stored as a fixed-length linked list of coordinates. Every x milliseconds the current position of the character is added at the front of the list and the last element is removed (the scent at this position got too weak to be noticed by the agents). It is important not to do this every frame: that way, the period of time that the character can be traced would depend on the current frame rate. Besides, this would be too much of an administrative overhead. Every time the agent attempts to smell the character, it has to be checked if the agent is within a certain distance of any point stored in the linked list. The maximum distance to the point might depend on the position of it in the linked list (a smaller maximum distance if the position is at the end of the list) to model the fading of the scent. The advantage of this approach is that the direction the character moved to is implicitly stored in the linked list – so if the agent has the abilities of a sniffer dog, it might start to follow the character based on his scent.
If the agent has to be aware of more than just the character's smell (for example, a creature might get alerted if it smells a dead body of a fellow creature even if it neither sees nor hears it), then again the nearby waymarkers can be used to store this information. Every waymarker needs a field for each kind of smell that can occur to indicate the smell's intensity. This information is updated every x milliseconds by something like:

```
wm.value_new =
max(0, (wm.value_old – declineFactor), (1 - (|wm.pos – source.pos| / maxDistance)))
```

In this example, the smell of an objects decreases linearly both in space and over time from 1 (maximum smell) towards 0 (no smell at all: the source is farer away than maxDistance and has not been around since "*x * (1 / declineFactor)*" milliseconds).

Vision is by far the most difficult sense to model as it has to be most accurate (mistakes made in the sense of seeing are more obvious to the player than in the sense of hearing and smelling). Two different approaches exist: object-based and image-based techniques. Image-based techniques are simple in principle and more accurate, but very computation intensive, complex in the implementation and therefore not used very often: basically it does the whole rendering process from the agent's point of view, without textures and pixel shaders. Only the character's object is marked with a special color. If there are areas of that color in the rendered image, then the agent can see (parts of) the character. If there are none, then the character is either not in the field of vision, or occluded by objects, hence the agent cannot see him. Doing this whole process for each agent is obviously too expensive, and is highly complex as it usually involves some special interaction with the 3D-accelerator of the graphics card.
Therefore, in most cases object-based techniques are preferable. The basic idea is to cast one or more rays from the agent's eyes in the direction of the character and check if they are within the visibility-range of the agent and what objects they collide first with (using the collision detection algorithms of chapter 5). If any ray is within the visibility-range and hits the character first (and not a tree, a rock or some other occluding object), then the agent can see the character. A set of six rays might be a good compromise between "not exactly enough" and "too much computation": one ray to each of the four limbs of the character, one to the head, and one somewhere to the center, around the stomach.

The introduced perception models are greatly simplified over the perception real humans or animals have. However it appears to be accurate enough for most 3D-action-games at the current time. In *Halo* (2003), for example, the hearing capabilities are using similar mechanisms

as described above, and for gameplay's sake an additional portion of unrealism is intentionally included: the agents can only see the player if the player can see the agent [4].
101 Dalmartians uses the simple hearing model (*notice = ((loudness / distance²) > threshold)*). To decide if an agent can see the character, the ray-casting technique described above is used, however with one serious problem that could not be resolved within the project time: only objects modeled as OBBs for collision detection can occlude the vision to the character, however not the ground generated by the heightmaps.

7.2. Decision making

Decision making is the core element of artificial intelligence. Many models are developed so that agents can decide, based on the percepts, knowledge, memories and experiences, what to do next. The approach used in games is different from normal behavior modeling: while the goal in behavior modeling usually is to create a decision making process that comes as close to the way humans think as possible, the goal for games is to make the agents "appear to be intelligent, rather than actually intelligent" [4]. The predominant decision making model are therefore finite state machines.

In a finite state machine, the agent is always in exactly one of a fixed set of states (like Attacking, Idle, Eating, Die). From each state, transitions to other states are possible, once a given precondition is fulfilled. The two common ways to describe these states and transitions are state diagrams and state transition tables [10].
As a result of this very simple model, the agents are solely reactive and not planning their future actions in advance. The big advantage of finite state machines is that they are very easy to design, implement, and that flaws and errors can easily be located.
Several improvements have been made to the simple finite state machines to improve the flexibility of agents while keeping the simple concept. In Fuzzy State Machines, for example, the agent can have several states at the same time, each of them to a certain degree. The model used in 101 Dalmartians to enable complex behaviors like "Look for food" is the Hierarchical Finite State Machine. In that one, each top-level-state can be divided into several sub-states, recursively. So every state can be another Finite State Machine. For example, Pongo's "Providing Alpha-Female with food"-state consists of five sub-states: Init, Searching for food, Collect food, Return to the nest, Give food to alpha-female. Once Pongo is in this top-level-state, that does not mean all sub-states will be reached. If Pongo is already at the food to begin with, the "Going to food" will be skipped. More importantly, he can leave the "Providing food"-state before reaching the final sub-state, for example if he sees the main character (new state: Attack) or is getting too hungry himself (new state: Looking for food). Some states cannot be interrupted like this: if the spawns are in the state "evolving", they are in their egg and will remain there even if the character attacks.

Theoretically, a hierarchical finite state machine can be seen as a plain finite state machine. In that case, every transition that leads to the top-state will be directed to the "Init"-sub-state, and every transition that is going away from the top-state has to be added to every single sub-state. However this leads to very complex models, so the modularization by using a hierarchical finite state machine is preferable.

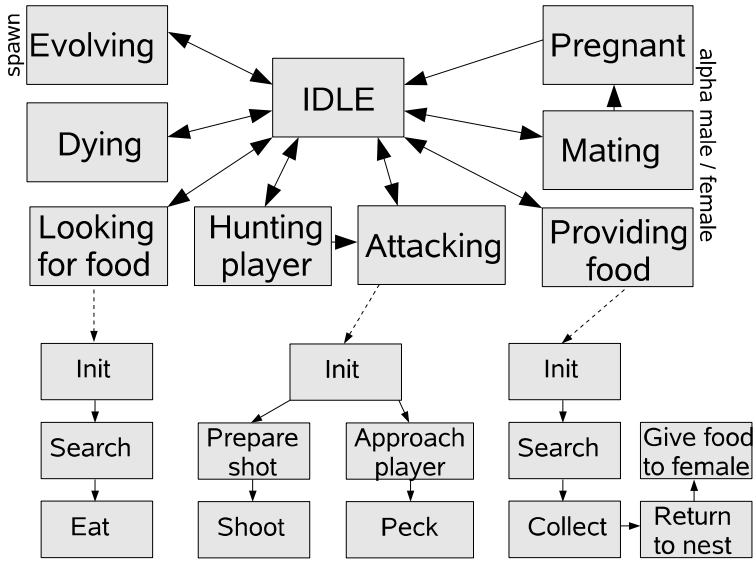

Fig. 7.1: The top-level-states of 101 Dalmartian's creatures are shown on the upper part. The lower part shows the sub-states of three major top-level-states.

Some games have been created that are using more sophisticated models. For example, the 3D-shooter “No One Lives Forever 2” (2002) uses a simple goal-directed system [4], the agents of the racing game “Colin McRae Rally 2” are using neuronal networks to drive the cars well on different surfaces [5] and the famous “Creatures”-games are based on neuronal networks and genetic algorithms. However, especially in 3D-shooting games, the most wide-spread model is still the “Finite State Machine” and models derived from it, used in popular games like Quake, Unreal and Half-Life [2].

7.3. Steering

When agents want to go to another place in the level, they will usually not have to plan the whole way for themselves but can rely on structures provided by waymarkers. The path that result from the path finding algorithms introduced in chapter 6 will help them to avoid big *static* objects that were taken into account when designing the waymarker structures. However, collisions might still occur with dynamic objects (like other agents or the main character) and small static objects. So the agents need a way to avoid objects locally.

A common technique that works well in most cases (however does not scale very well) is to use repulsion forces that are emitted axially from every object. First of all, the vector of the direction the agents wants to go is calculated. Then, the repulsion force vectors from nearby objects are added up. The closer the objects are to each other, the stronger the repulsion force. So if two agents are going in opposite directions and are about to slightly touch each other, the repulsion force will make them avoid each other, resulting in smooth turns to the side. The repulsion force has to be optional, though, as there are cases in which collision between the agents and other objects are desired (for example when a creature wants to ram or peck Cruella, or Pongo and Perdita are mating).

The repulsion force works if the agents are *not* going along the exactly same line. If that is the case, the repulsion force is exactly opposite to the desired walking direction and thus does not make them step aside. Unfortunately this happens frequently, as the agents will follow the paths given by the waymarkers most of the time. So this case needs a special treatment. In "101 Dalmartians", the following workaround is used: if two agents are getting close to each other and the resulting vector and the vector of the originally intended movement make up an angle less than 10°, then another vector pointing to the left of the agent is added to it. By this, they will avoid each other by going to the left side from their own perspective (the decision to go to the left side was made according to the left-hand traffic system in Singapore). As a last resort, if this does not help either (which can happen if more than only two objects are involved), the agents just turn around if they notice they will not get any further for a while and go back to the last waymarker.

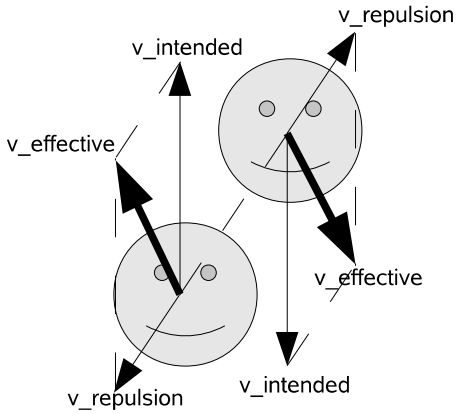

Fig. 7.2: Two creatures are approaching each other and are about to collide. The repulsion force makes each one go to its own left and therefore avoid the collision.

This system works sufficiently well in most cases if there are not too many agents running around and there are not many narrow passages. However, if many objects are involved, then this systems tends to break down. The simple approach is to try to avoid these situations, by making the agents choose paths that are not used by other agents. This could be done in the path finding algorithm, by including it in the cost function (if the paths are computed during runtime). But then again, this might not be the desired behavior, if the agents are supposed to stick together in order to be stronger in combat as a group.

To solve this problem, real local planning is needed, which leads to very complex search algorithms which would be too complex for the scope of this work.

7.4. Memory

In most action-games, the enemies will not need much own memory, if any at all. In a typical 3D-shooter, there are two things an agent might want to remember: the position it saw the character the last time, and the location of power-ups (food, in the case of 101 Dalmartians). Implementing the memory is straight-forward – a simple array is usually enough. The one thing that needs to be taken care of is the fact, that the reality of the level might have changed in comparison to the memories an agent has. The character might have went to another place, and the power-ups the agent saw might have been used by the character or another agent. In 101 Dalmartians, the food can change the location – the character can kick it to another place. So the object still exists, but the new position of the object does not match the memories of the agents anymore.
It is questionable if this kind of realism really increases the fun of the game. During the design of *Halo*, the creators noticed that some intentionally added realism was interpreted as a bug in the AI by beta-testers. For example, after seeing the character, the agents needed a short time to recognize that the character is the enemy and not a friendly unit. While this is realistic, many players complained about this "bug" during beta-tests [3]. In the same way, if a player sees that an agent wants to fetch a power-up at a place where there is none, this might be interpreted as a bug. For 101 Dalmartians, where a big emphasis was put on creating realistic agents in the project requirements, it was decided to include this source of realistic mistakes.
Once the agent sees a fruit, a structure we named "ghost" is created in the memory of the agent. It contains the position and the collision properties of the original object. For subsequent

visibility checks, it will not only be checked what fruits the agent actually *can* see, but also which fruits the agent *could* see according to the ghost-information in the memory. If it could see a fruit at a specific position but there is none, then the ghost is deleted from the memory. This way, even if the agent has obsolete information about food, it does not have to go all the way to the old position. Once it *sees* there is not fruit anymore, it will look for another one.

7.5. On- and Off-Screen behavior

Many parts of the agent's behavior is very computation intensive. For each rendered frame, the collision tests, steering, visibility checks and so on have to be performed, which might slow down the game too much if the level is big and many agents are wandering around.
One common way to solve this problem is to use different behavior-routines – one computation-intensive routine that does all these checks, and one routine that does as little computation as possible. The latter one is used if the agent is far off-screen. So while this results in much less realistic behavior of the agent, the player will hardly notice that as he cannot see the agent at that time.
Many simplifications for off-screen behavior are feasible:

- The visibility checks can be done much simpler. Instead of casting rays from the agents to the objects and checking if they collide with an occluder first, all objects within the view cone of the agent and a certain distance are seen. Or to make it even simpler, all objects within a certain distance can be seen, saving the check of the field of view.
- No collision tests need to be done when the agent is following the predefined paths. When two agents meet each other along the way, they can just walk through each other. And no big static objects are on the predefined paths anyway.
- Updating the animation while walking can be omitted. If the rendering engine does a good job, the rendering information will not be sent to the graphics card, otherwise it should be manually set as invisible.
- An even more aggressive approach is not to move the agents the normal way at all but to “warp” them from one waymarker to another. When the agent is at one waymarker, the distance and walking time to the next waymarker it wants to go to is estimated. A counter is set to the walking time, and once the counter is down to zero, the agent is warped to the target. This way, all the calculation how far an agent can walk from one frame to the next one is saved.

The big disadvantage of this method is that the transitions between the two behavioral routines (especially from off- to on-screen) is highly complex. While collisions are simply ignored in the off-screen mode, the on-screen-mode must be collision-free. So on the transition from off- to on-screen-mode, the position of the agents might need to be interpolated (when using “warping” for walking), and relocated to avoid collision.
As the transition is also computation intensive, it should be tried to have as little transitions as possible – better let the agents be a bit longer on-screen than to let them be off-screen as soon as possible only to switch back few frames later. The simplest way is to delay the transition to off-screen a bit. Once the agent is far enough for some seconds already, it is less likely that it will be on-screen soon again.

In 101 Dalmartians, the mentioned methods of off-screen-behavior were implemented, but not used in the final version as the transition from off- to on-screen could not be implemented fully consistently in time.

8. Implementation

8.1. Toolkits and SDKs

The core engine used for “101 Dalmartians” is OGRE, an open-source (GNU Lesser General Public License) 3D graphics engine. It is written in C++, however a wrapper for Java [link 7] and a port to C# [link 8] also exists. It is important to point out that Ogre is not a game engine, so it does not deliver functions for building artificial intelligence, game physics, sound, and only rudimentary functions for handling user input and collision detection.
Ogre uses a scene-based approach, in which a tree-shaped scene-graph is constructed and objects like the logical camera or rendering objects are attached to SceneNodes.
One of the most important features of Ogre is to provide an abstract class “RenderSystem”, which is implemented for different kinds of low-level rendering systems. Implementations for OpenGL and DirectX are provided, so it is rather simple to write programs that can be compiled for Windows, Linux and Mac OSX.
ResourceManager objects ease the management of graphic objects, especially textures and meshes. These objects are referenced by name rather than by the filename – a configuration file usually called “resources.cfg” provides the mapping between these logical names and the physical location of the files on the disc. Therefore, the program does not have to be recompiled when a filename changes.
The most important object in Ogre is the “SceneManager”, again an abstract class with different implementations. It manages all the SceneNodes, cameras, lights and materials and does most of the communication with the RenderSystem. Some of the implementation can use heuristics to find objects that are not on the screen. These objects are not sent to the rendering system, which greatly improves the performance of the game. For example, big in-door levels can benefit from the “BspSceneManager” that uses a BSP-tree to organize the static world. “101 Dalmartians” uses the “TerrainSceneManager”, that provides functions to generate the ground of the level from bitmap heightmaps.
The scene itself is managed in a tree-structure consisting of an arbitrary number of “SceneNodes”. The first node is called “RootSceneNode”. Every node can have several child-nodes. All objects to be rendered, like the meshes, the light sources, but also the camera, are attached to a SceneNode. The big advantage of this system comes with the fact that the position and orientation of each node is stored relatively to its parent node. So it is possible to move several objects that belong together by change the position of only one SceneNode.
Once the “startRendering”-function of the Root-object is called, Ogre starts the rendering and takes over the command over the program. It will try to render as many frames as possible with no delay between the frames (which is usually the way to go in games). The game loop, the core function of the game, is an implementation of the “frameStarted”-function of the “FrameListener”, which is called every frame. The game loops gets the information how much time has passed since its last call, it can query the status of the keyboard and the mouse and update the SceneManager. Once the “frameStarted”-function returns a “false” value, Ogre ends the game.

To include sound effects and a background music to the game, the “Fmod EX” [link: 9] audio library is used. Fmod EX is a very powerful cross-platform, multi-format engine that has a 3D- and a 2D-mode. In 3D-mode, many sound sources (like creatures) and one listener (the main character) can be defined, including their position and velocity. Several 3D-effects can be achieved (like doppler shifts when the sound source or the listener is moving). “101 Dalmartians” uses the the 2D-mode, where sounds are simply played without any further information about the source's position. Only the background music and one sound is played at the same time in this game.

8.2. Tools for creating the level

The structure of the level is not hard-coded, but given by a set of input files. The core input files are:

- Terrain.cfg is loaded by the TerrainSceneManager. It provides the heightmap and the textures for the ground.
- Level1.cfg provides the locations of static objects (rocks, trees, archs), food, the initial creature population and the main character in the level.
- Regions.txt defines the rectangular areas the level is divided up into, the waymarkers and boundary points, and the connections between the waymarkers.
- Interregion.txt and Intraregion.txt are the precomputed cache-files used for path planning.

Except for Terrain.cfg, all files are created by tools that were written specifically for this game. The most essential tool is the in-game level designer. It is called by pressing “F12” in the “cheat”-mode (triggered by the enter/return-key). From a bird-perspective, the static objects, creatures, food and waymarkers (represented by a cross) can be placed. A special mode exists for creating the connections between waymarkers (represented by white lines between the waymarkers). The rectangular regions and the initial position of the character has to be entered manually into the files, as it would have been more effort to create tools for doing this.
If the setup of the waymarkers and their connections has been altered, then another tool has to be called - “connection.exe” creates interregion.txt and interregion.txt using the algorithms described in chapter 6.2.
Another tool was written to create the OBBs assigned to each object. The “OBB designer” allows to create a box in the game and modify it along the 4 degrees of freedom (x/y/z-axis and y-rotation; see chapter 5.4). So, one box after another, a bounding volume structure for an object is created which will be used for the collision detection. Due to the time constraints on the project, this designer had to be done “quick & dirty” - and indeed it is dirty. The mesh that the OBBs are created for is hard-coded, so after creating all the OBBs, the program has to be modified and recompiled to select a new mesh. The resulting OBBs are hard-coded into the program, too. For bigger projects with many different kind of meshes, this would be unpractical, but in “101 Dalmartians”, only around a dozen meshes are used and around 50 OBBs had to be created, so it was the fastest way to do it like this.

8.3. Class Structures

In the given UML-diagram (fig. 8.1) of the main classes used in the code, the program is divided into three parts: At the left are the “Ogre Standard classes”, which are provided by the Ogre API. No modifications were made to these classes, except for inheritance. The middle part is the “game loop” part, where the character-control, the creature's actions and mostly everything else dealing with the actual game playing happens. In the “abstract world”, there are classes for collision detection, path finding and knowledge representation. The separation between the “game loop” and the “abstract world” part may be a bit artificial, however it makes clear that from a conceptional point of view, the game physics and the rendering are two completely different tasks that do not influence each other.

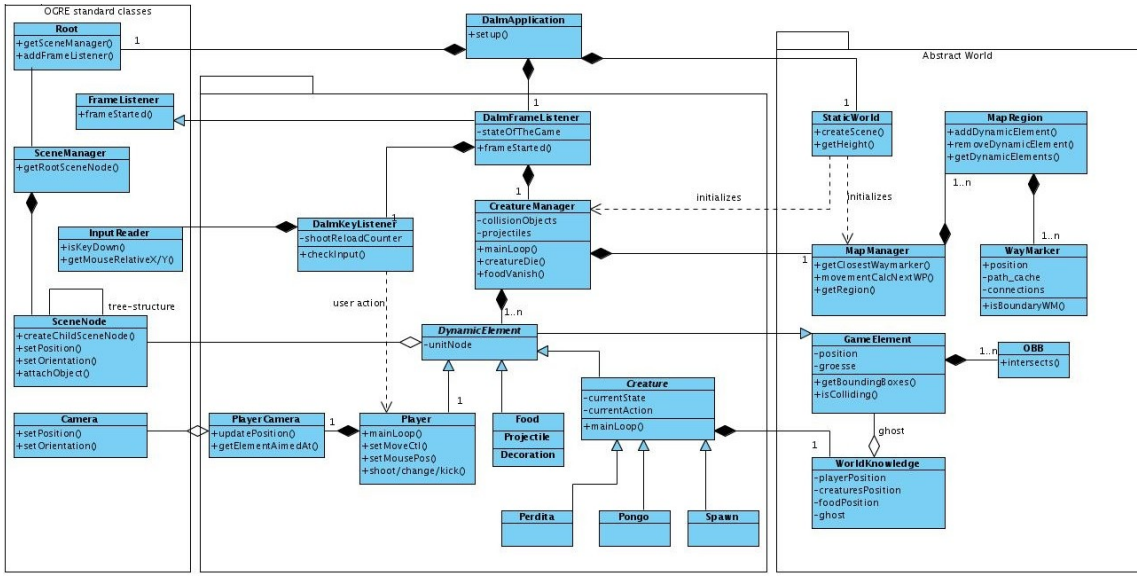

Fig. 8.1: Class diagram of 101 Dalmartians

A brief description of the main classes:

- DalmApplication is the “root class” of the whole program. It is called by the operating system, does all the initialization of the main underlying classes, and finally calls the startRendering()-Function of Ogre's Root class. After this point, DalmApplication will be passive until the game ends.
- DalmFrameListener contains the main Game Loop. It handles the different game modes – like the playing mode, the help screen, the designer tools, and so on. In the playing mode, its main functions are to call the input handler DalmKeyListener and the game loop of the CreatureManager.
- CreatureManager actually does much more than its name sounds like. It handles not only the creatures, but also the character, the food, and the projectiles. In addition to that, it keeps record about the static object in the world. It owns the “MapManager”, which divides the level into logical areas, keeps track about which objects are in which area and manages the waymarkers and path finding. The collision detection routines rely on the information provided by these two objects.
- The Player-object is connected to two other objects besides the CreatureManager: first, it receives commands from the DalmKeyListener, like “shoot”, “turn to the left” and so on. Secondly, it is connected to the PlayerCamera. In normal game mode, the Camera's position mainly depends on the character's coordinates and orientation, however in some special modes (for example in the designer mode) it can be detached from the Player-object and be controlled by the DalmKeyListener directly. If the character shoots, the information about the creature aimed at is provided by the PlayerCamera to the Player-object. This may sound strange at first, but comes from the gameplay design: The player aims using the crosshair that is fixed at the monitor's center. The camera's position depends on the character's position, but to avoid collision and in order to look up and down, it is flexible to a certain amount. Thus the Player-object does not know by itself where the camera exactly is.
- DynamicElement and GameElement are nearly identical. GameElement is every object in the level that has a shape and position. DynamicElement is a GameElement that physically exists in the game world and therefore has a mesh attached. All the actors in the game are DymanicElements as they exist in the physical (rendered) world and in the abstract world.

- The Creature-objects, including the Spawns, Pongo and Perdita, are basically one big game loop controlling the state machine. As complex as the state machine is, the underlying design quite simple. Every Creature-object has one attached WorldKnowledge-object, that is basically a container for various information the creature knows about the world. The WorldKnowledge can contain GameElements that represent the shape and position of the objects the creature remembers (the “ghost” mentioned in chapter 7.4). This structure has the advantage that the same functions used to check if a creature can see a real object can be used to check if the creature could see an object if it still was at the same position as before. If it cannot see it for real, then the creature can reason that the object has moved or vanished and update its memory accordingly.

At this point, it should be mentioned once more, that all of the game design had to be done on-the-fly under heavy time constraints, while being somewhat unexperienced in game design at the beginning. The presented class diagram is therefore a highly idealized version of the real source code, and even so, some weaknesses of the game design are apparent (the naming of the GameElements and DynamicElements is highly counter-intuitive, for example).
One more thing to point out is that some core objects like the SceneManager, CreatureManager, MapManager and StaticWorld are basically globally visible. As nearly every other object needs functions provided by them, nearly every object has a reference to them. This is implemented by passing the references when constructing the objects. A more elegant way would have been to use the “Singleton”-construct of Ogre that makes objects that are instantiated only once globally available.

9. Discussion

To conclude this work, some experiences made during the project and the current state of the game will be presented.

9.1. Lessons leaned

If the goal of the project had only been “design a good game”, then the biggest mistake would have been not to use a real game engine. Of course, given the academical background of the project, using an existing game engine was out of question. However, one of the most important points learned was to appreciate the work of game engines. While the core concepts of the game physics seem to be easy to understand, a nearly endless amount of issues arise to make things work smoothly. Even an apparently easy task like making an agent walk from one point to another one can be quite bothersome, finding a good route to that point using waymarkers, over avoiding the collisions with other agents, to handling the correct walking speed once a slope is involved. And all of this has to work before one can event start to think about higher-level artificial intelligence. So if having a working game had been the main intention of the project, using an existing game engine instead of writing a new one from scratch would have been the better decision – but even then, the three months of time probably would not have been enough to implement all the ideas of the initial concept. Creating a 3D-game just takes time.

Putting the game physics part aside, it has to be pointed out that the workload and importance of the graphics cannot be overestimated, even for a rather small project like “101 Dalmartians”. While this document is mainly about the underlying algorithms, the graphics and animations still was one of the most time-consumptive tasks of the project. For every team of coders trying to create a game, it will be crucial point to find enough people good with graphics.
Luckily, both Gilles and Cedrik were quite talented with graphics, so even when the game was made only by computer science students without the help of any “professional” designers, the game looks quite decent given the circumstances.
Of course, the graphical quality cannot match the quality of commercial games. However, watching the reactions of “normal people” who are not involved with any kind of software developing on the game, it seems that this “fan factor” does not influence the expectations on a game very much - a game *will* be compared to commercially available games, no matter the difference in developing time, experience and budget.

An interesting “mistake” that some people pointed out was the problem to identify with the main character of the game. The problem they had was the fact that you see Cruella only from the back – so in the end, some people did not even find out if she was male or female. For 3rd-person-games, it seems to be important to see the main character from the front side from time to time.

9.2. Current state of the game

Currently, the state of the game could be described by “playable”. It worked well on the lecture-internal demonstration at the end of the project. It did have some compatibility-related issues – some of them have been removed after the project was over, some of them have not. The game worked at approximately two third of all tested systems.

The game is quite enjoyable for a while – it has sweet graphics, funny sounds, and a game element not to be underestimated: the “kick”. Basically, the kick serves no real purpose, which was why I myself objected to spending a lot of time into coding it at first. Most of the kick was therefore coded by Gilles, whose idea the kick was to begin with. In the end, he was right: watching other people playing the game, we realized, that most of them found the kick to be the most entertaining part of the game, running around in the world, trying to kick everything from creatures (possible), fruits (possible) to 20m-high rocks (not possible). For a game, it seems to be vital to have an “unique sales factor” like this in order to be really entertaining. Lesson learned.

Once one has a closer look at the game, a lot of rough edges can be found. Some of the most annoying bugs are:

- A nasty graphical bug appears once shadows are activated. The cause was not found yet, it is probably something between Maya and Ogre. As several bugfix releases of Ogre were released in the meantime and none of them mentioned this problem, it is probably some faulty settings at the exporter-plugin for Maya.
- The steering of the creatures is far from perfect, resulting in creatures colliding with each others from time to time or even get stuck completely.
- Sometimes the creatures have problems walking up or down the slope, making them walk backwards – a strange effect we dubbed “the moonwalk” that we could not completely resolve yet.
- Some collisions are not very accurate yet – especially the nest of Perdita. The character sometimes can walk completely into the rocks. The problem does not lie in the algorithms used – the problem was just that the time was not enough to define all the OBBs accurately. As the nest is the most complex object in the game and only appears at the end of the game, where the player is probably busy having his showdown with the alpha male and female, not much time was spent into the nest.
- Many other details – most of them were mentioned earlier in this document.

Fig. 9.1: A still unresolved graphical bug at the edges of the gray stones.

However, there is no real work going on at the project anymore. After the project, few of the most annoying bugs and mistakes were resolved, but there is no intention to commercially release this game. For that reason, it was decided to release the code under some Open Source licence, downloadable at [18].

References

1) Harald Bögeholz. Kollisionskurs – Physikbeschleunigung will Spielewelten bereichern. In: c't, 10/2006, p. 108.
2) Jason Brownlee. A Practical Analysis of FSM within the domain of first-person shooter (FPS) computer game. October 2006. http://ai-depot.com/FiniteStateMachines/FSM-Practical.html (included in the CD)
3) Chris Butcher, Jaime Griesemer. The Illusion of Intelligence - The Integration of AI and Level Design in Halo. In: Game Developer Converence, San Francisco, March 23, 2002. Slides: http://halo.bungie.org/misc/gdc.2002.haloai/talk.html?page=19 (included in the CD)
4) David E. Diller, William Ferguson, Alice M. Leung, Brett Benyo, Dennis Foley. Behavor Modeling in Commercial Games. 2004. http://www.sisostds.org/index.php?tg=fileman&idx=get&id=2&gr=Y&path=CGF-BR%2F13th+CGF-BR%2F13th+CGF-BR+Papers+and+Presentations&file=04-BRIMS-079.pdf (PDF-file included in the CD)
5) Stephan Ehrmann. Mensch-Maschine-Duell – Die Fortschritte der Künstlichen Intelligenz in Spielen. In: c't, 9/2006, p. 96.
6) Miguel Gomez. An OBB-Line Segment Test. October 18, 1999. http://www.gamasutra.com/features/19991018/Gomez_6.htm (included in the CD).
7) António Ramires Fernandes. 3D-Maths for CG – Ray-Sphere Intersection. October 2006. http://www.lighthouse3d.com/opengl/maths/index.php?raysphereint
8) Jeffrey Mahovsky, Brian Wyvill. Fast Ray-Axis Aligned Bounding Box Overlap Test with Plücker Coordinates. In: The Journal of Graphics Tools Vol. 9 No. 1, AK Peters Ltd, Wellesley, Mass, 2004. p. 35-46. (included in the CD)
9) Katie Salen, Eric Zimmerman. Rules of Play – Game Design Fundamentals. The MIT Press, Crambridge, 2004. p. 453-455.
10) Wikipedia. Finite State Machine. October 2006. http://en.wikipedia.org/wiki/Finite_state_machine
11) Wikipedia. One Hundred and One Dalmatians. October 2006. http://en.wikipedia.org/wiki/One_Hundred_and_One_Dalmatians

Links

1) http://www.nus.edu.sg
2) http://disney.go.com/disneyvideos/animatedfilms/101/
3) http://www.ogre3d.org
4) http://www.autodesk.de/maya
5) http://www.ogre3d.org/index.php?option=com_remository&Itemid=57&func=fileinfo&filecatid=19&parent=category
6) http://ftp3.ru.freebsd.org/pub/sourceforge/o/og/ogre/
7) http://www.ogre4j.org/
8) http://axiomengine.sourceforge.net/
9) http://www.fmod.org/

Figures