Charles University in Prague
Faculty of Mathematics and Physics

BACHELOR THESIS

Michal Brabec

Skeletal Animations in Real-time 3D Graphics

Department of Distributed and Dependable Systems

Supervisor of the bachelor thesis: Mgr. Pavel Ježek

Study programme: Informatika
Specialization: Programování

Prague 2011

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

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

In…......... date................

Název práce: Skeletal Animations in Real-time 3D Graphics

Autor: Michal Brabec

Katedra / Ústav: Katedra distribuovaných a spolehlivých systémů

Vedoucí bakalářské práce: Mgr. Pavel Ježek

Abstrakt: Skeletální animace je velmi efektivní animační technika, která se používá ve videohrách a filmech k animaci lidí, zvířat a dalších složitých modelů. Tato práce se zabývá mnoha problémy spojenými s implementací skeletálních animací a nabízí jednoduché a efektivní řešení těch nejdůležitějších. Práce obsahuje veškeré znalosti nutné k pochopení a použití této důležité animační metody protože vysvětluje všechny operace a výpočty nezbytné pro transformaci modelu podle jeho kostry. Součástí této práce je kompletní implementace skeletálních animací a tato implementace je postavena na konceptech prezentovaných v hlavním textu práce. Hlavní aplikací je editor skeletálních animací, který používá prezentovanou implementaci skeletálních animací a umožňuje uživatelům měnit animace 3D modelů. Animační editor je grafická aplikace s komplexním uživatelským rozhraním a jeho konstrukce je vysvětlena velmi podrobně.

Klíčová slova: skeletální, animace, DirectX

Title: Skeletal Animations in Real-time 3D Graphics

Author: Michal Brabec

Department / Institute: Department of Distributed and Dependable Systems

Supervisor of the bachelor thesis: Mgr. Pavel Ježek

Abstract: Skeletal animation is an effective technique for the animation of humans and animals, and it is used in most video games to animate complex 3D models. This thesis addresses many problems of the implementation of skeletal animations and it presents an effective solution to the most important ones. It contains all the knowledge necessary to use this important animation method, because it explains all the calculations required to transform a model by its skeleton, and then this thesis presents a complete implementation that uses all the concepts explained there. Build on the skeletal animations, the animation editor is the main application of the thesis, and it allows users to modify animations of 3D models. The animation editor is a graphical application with complex user interface and its construction is explained in a great detail.

Keywords: skeletal, animations, DirectX

Skeletal Animations in Real-time 3D Graphics

Project Dragonfly

Contents

Skeletal Animations in Real-time 3D Graphics....4
1. Introduction....6
 1.1. Model animation basics....6
 1.2. Basic animation techniques....6
 1.3. Morphing....6
 1.4. Key-frame animation....7
 1.5. Skeletal animation....7
 1.6. Parallel programming in graphical applications....8
 1.7. Problem statement....9
 1.8. Project goals....9
 1.9. Project goals summary....10
 1.10. Minor goals....10
2. Analysis....12
 2.1. Skeletal animation....12
  2.1.1. Overview....12
  2.1.2. Data representation....12
  2.1.3. Skeleton implementation....14
  2.1.4. Skin implementation....14
  2.1.5. Animation implementation and used computations....14
  2.1.6. vertex properties and materials....16
  2.1.7. Extensions and additional functionality....17
 2.2. Graphical framework and animation editor....18
  2.2.2. Extension of the animated model....18
  2.2.3. Parallel environment and synchronization....20
  2.2.4. User interface and input management....22
  2.2.5. Direct3D environment and graphical device....22
  2.2.6. Environment construction and maintenance....23
  2.2.7. Error management and warning report....24
3. Implementation....26
 3.1. Skeletal animations....27
  3.1.1. Animation library....27
  3.1.2. Basic structures 28
  3.1.3. BaseModel interface....30
  3.1.4. Animation class....31
  3.1.5. AnimatedModel class....31
 3.2. Animation editor....32
  3.2.1. Technical specification....32
  3.2.2. Module design....32
  3.2.3. Module description....33
  3.2.4. Run-time design....34
  3.2.5. Timer module....39

3.2.6. Threading module....................39
3.2.7. Windows module....................41
3.2.8. DirectX module....................43
3.2.9. Control module....................45
3.2.10. Project module....................48
4. Comparison....................51
4.1. Main graphical libraries....................51
4.2. Library support for skeletal animations....................51
4.3. Other open implementations....................53
5. Conclusion....................54
5.1. Results....................54
6. Appendix A....................55
List of Figures....................56
List of Tables....................56
Bibliography....................57

1. Introduction

Computer animation is a wide area of computer graphics, that is present in almost every graphical application. This project is primarily aimed at the animation of 3D[1] models and the main technique discussed here is a key-frame based skeletal animation of 3D triangle meshes[2]. It is a special method used in movies and video games for animation of humans and animals. Inanimate objects can be also animated this way but there often exist more effective approaches. This text explains the main ideas behind this technique, and then it describes one complete implementation. It presents used solutions and any other possibilities are discussed and compared.

1.1. Model animation basics

A model is a simple triangle mesh and in this project, it does not matter what it looks like, it is just a list of triangles. This text assumes that the models are constructed from a vertex buffer[3] and an index buffer[4]. A pose of such model is its exact shape – position and size of each triangle. Each pose is technically a different model but they are called poses because they look alike, and they are used together. There are many ways to animate such a model. The first one is to prepare many versions of the model, each in a slightly different pose, then just render these models in a loop. This is a very simple method, usually used in 2D[5] graphics, but it can be very inefficient for the purpose of 3D graphics. The 3D data are much bigger then the 2D data, and to load and store them can be very costly, but the biggest problem of this method is that the model must be loaded to the video memory so it can be rendered effectively. The video memory is usually smaller then the main memory and if the model is constantly changed then it must be repeatedly moved from the main memory to the video memory. However, this method can be used for simple models which are small and their transfer won't affect the efficiency, but it should not be used for complex models.

1.2. Basic animation techniques

Loading and rendering every pose of a model can be costly, but there is another possibility to animate 3D models. The model can be stored in a few, most important, poses and the rest of the animation is computed at run time. This approach is more efficient, it removes some memory requirements and adds some computational requirements. The computation is done for every vertex and there can be a lot of them, but the graphical hardware is designed to work with vertices very effectively. The computations are massively parallel and thus they should be very fast. In general, there are two techniques derived from this idea – morphing and key-frame animation.

1.3. Morphing

Morphing is a method mainly used to animate facial expressions. The main idea is simple: a model is stored in extreme poses, like angry and happy, and the result is computed from them based on a simple ratio. It is also possible to animate the changes in the expression by modifying the ratio

1 Three dimensional space.
2 Collection of triangles that define a shape of a polyhedral object in 3D computer graphics. More info can be found in [MSDN1].
3 A vertex buffer is used to store models vertices and their parameters. [MSDN2]
4 An index buffer stores the information about vertex organization. [MSDN3]
5 Two dimensional space.

gradually as the expression changes. This technique will not be used in this project and it is presented here only as an example of another animation method.

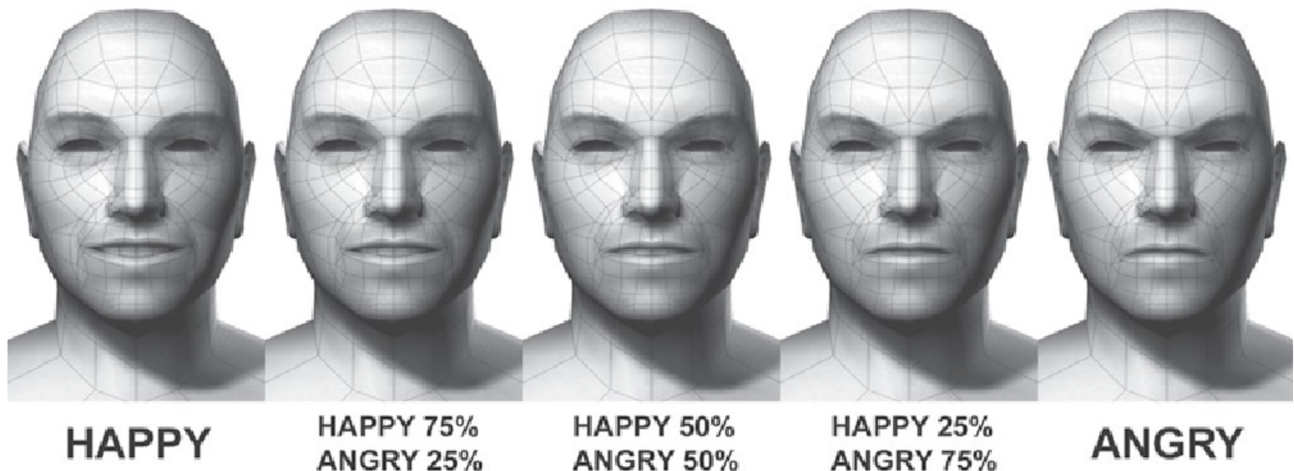

Figure 1: An example of morphing animation. The two extreme expressions are combined to create the intermediate ones. The image is from [Granberg09].

1.4. Key-frame animation

The key-frame animation is based on a simple idea, that a model is stored in a few important poses, and the intermediate poses are computed from them. Every single pose, so called key-frame, contains the information about a time when it should be used and the final pose is computed from the key-frames based on the actual time. First, it is necessary to compute the time, then select the source key-frames. There are many possible ways to compute the final pose, the closest key-frame can be used or many key-frames can be interpolated. This method is very effective especially if combined with other techniques and it is the main technique used in this project.

1.5. Skeletal animation

In the 1990s appeared a new method of model animation – a skeletal animation. It was a great revolution in character animation, mainly for video games. The method is based on the idea that models contain geometry and skeleton and only bones move and soft parts mimic the movement, thus the bones shape the models surface. This is very similar to reality for most vertebrate creatures, while the bones themselves are moved by muscles. Everything is moved by the bones, because they support the entire body of vertebrates.

In computer graphics, a model is stored as a standard 3D triangle mesh, then it has a skeleton and they are linked via a skinning information[6]. A skeleton is a hierarchy of bones that is organized like a tree[7] and bones are usually simple transformations, like matrices. Every bone defines a local transformation for all nearby vertices and it is relative to the parent bone – the parent bone defines the transformation origin. A bone is finally transformed by its offset to the original space and then all the vertices can be moved by it. The to-root transformation is computed recursively for each bone from its ancestors transformations.

6 Each vertex of the model contains an information about bones that influence its movement.

7 A graph without a circle sub-graph.

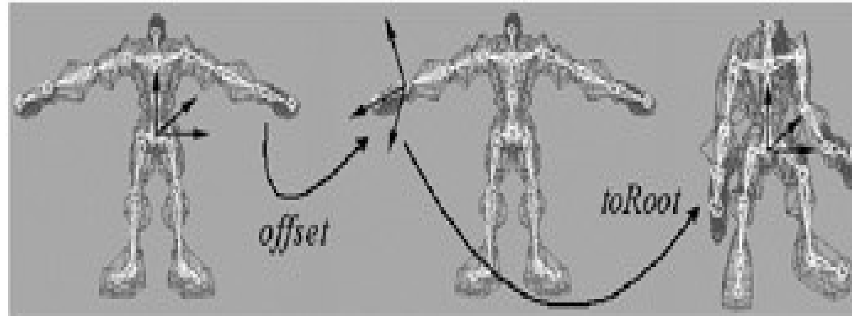

Figure 2: The skeleton binding. The image is from [Luna06].

Skeletal animation has many great advantages. First, only bones move and therefore they must be transferred to the video memory in every frame, but the geometry can be loaded permanently. This is very important, because the geometry is usually much bigger then the skeleton and loading big data to the video memory can hurt performance. Second, the animations are smaller because only changes in the bone hierarchy must be stored and everything else is computed from them. If the skeletal animation is combined with key-frames then the result is a compact and effective animation, that can be used for any model and movement. Third, if some models have the same or very similar skeleton then the animation can be used for all of them. This is very useful when there are models with a little different surface or just a different texture, because they can be all animated the same way. Such situation is very common in video games, when there are many humanoid models with similar skeleton and different surface – for example humans with different clothes or armor. The skeletal animation adds small computational requirements, but this means a few matrix multiplications per bone and a matrix multiplication per vertex and such computations are extremely fast thanks to the hardware acceleration.

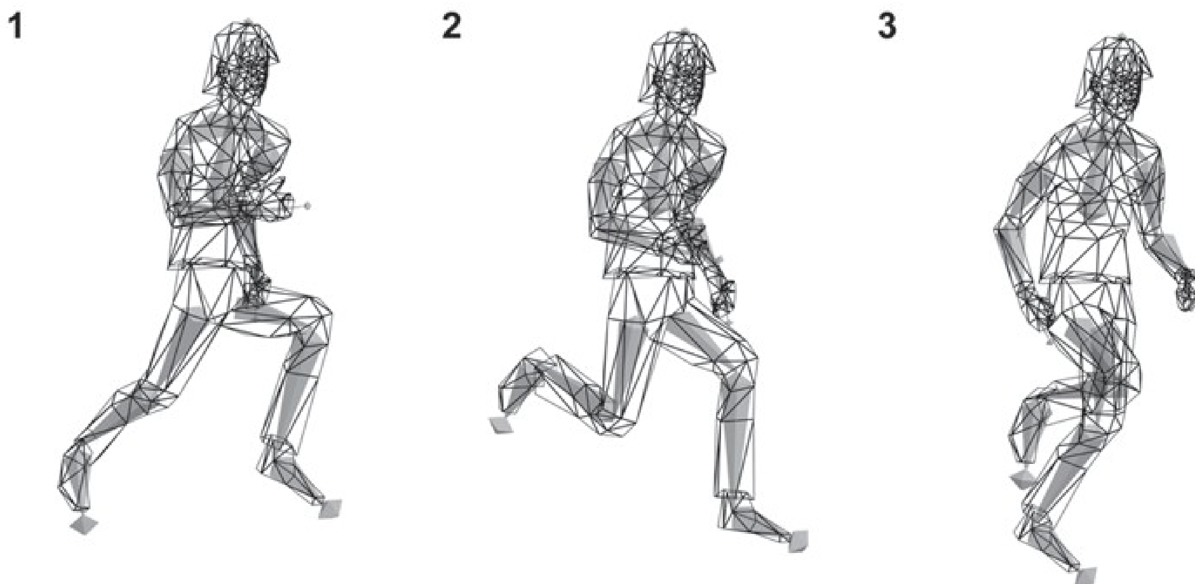

Figure 3: Three key-frames of a character and its skeleton. Additional poses are computed from these key-frames based on current time. The image is from [Granberg09].

1.6. Parallel programming in graphical applications

Graphical applications face some very specific problems and its general construction offers great opportunity to use parallel programming that may help applications performance and it makes separate subsystems more independent. The main reason is that the graphical applications usually

contain an input manager, that communicates with users, and then they use a complicated graphical engine. These two parts are very different and they can be separated because the graphical engine is simply waiting for commands and it mostly does not return any data. It just processes the commands and then it presents the graphics as its result. These two parts can be run in different threads.

Parallel programming has many specific problems and possibly the biggest is synchronization – safe communication between threads or processes. There are many ways how to handle the situation when two parallel tasks need to access the same part of a memory or some other resource. One solution is message synchronization based on the idea that if the thread encounters an important situation then it informs other threads using a short message. The messages are stored in each thread and they are delivered every few milliseconds. This method allows the threads to work without locks and waiting, when a thread wants to communicate with other threads it sends a message and then it resumes its work. The method has also some drawbacks. It requires very specific design of classes and functions, because they must be able to send and receive messages and the classes can really grow in size. Sometimes there can be a problem when a thread generates a message and then it needs some response, it can either wait – locked or it must have some mechanism for receiving responses. This method is not used very often, because it is more complicated to implement and the entire application must be designed so it can use it properly but it can be very effective in some applications.

1.7. Problem statement

Any complete implementation of skeletal animations must solve many problems and this project was designed to present simple and effective solutions for the most important ones. The most important problems are discussed here along with some examples.

There are usually many specific requirements, whenever an application uses skeletal animations and it must be possible to modify the functionality according to the specific needs of the actual application. For example, when a game allows characters to carry weapons, then the implementation must be slightly modified, so it can render the carried weapons correctly. On the other hand, the actual implementation must be complete and reliable, because its users usually want only to add some specific functionality and they do not want to implement parts of the actual skeletal animation.

There are always many problems during the development of any application and it helps if the implementation is as accessible as possible. The programmers should be able to check and control every part, because they must be able to observe all the changes in the animations so they can find all problems with their models and other parts of the application. This does not necessarily mean bugs or errors, but it usually refers to specific graphical effects and anomalies.

Many programmers, that are new to the skeletal animations, are confused and they should be able to check all the important parts of the implementation to see how it work. Programmers are usually not sure how the separate subsystems work and the entire solution must be open so they can make sure that everything is really done the same way as it is presented in the text. Therefore, the benefit of an open and extensible design is possible educational use, because programmers can inspect every part and that should help them to understand the principles behind this important animation technique.

1.8. Project goals

This project is aimed at the implementation of skeletal animations and it will present a complete implementation of this animation technique. The project is called Dragonfly.

The project will divided in two main parts:

1. The first part will be an implementation of skeletal animations designed as a separate module that will contain all the necessary classes and it will not depend on any other parts of the project so it could be used in other projects as a simple animation library.
2. The second part will be an animation editor, that will use the animation library to present its functionality and possible use. The editor will utilize messages for synchronization and it will implement a complete graphical framework and user interface. It will be called Dragonfly, same as the project itself.

The skeletal animation should be implemented completely, using only basic library functions, like matrix multiplication. This allows users to examine every part and they can either change it for their own needs or they can just look how it is implemented. If some parts are implemented using a complex library functionality then the user may not be able to change them or even see how they work. The implementation should be therefore open and without any hidden functionality and it must be scalable and extensible, for better use. At last, it must work well with the hardware for maximal acceleration, because the performance could be terrible without it. This project is centered around the basics and it will present solution to the main problems and possible approaches for the rest.

The message-based synchronization is very different from the standard lock-based one. It has some advantages and some disadvantages, but this project will mainly try to decide if this method is suitable for graphical applications. These application are specific in their requirements, because they have two main parts – the complex graphical interface and other parts for communication with users or the OS environment. The separate parts does not need synchronization very often, they must sometimes just inform the others about some changes, and that is perfect use for the messages.

The last goal of the project is to implement an animation editor that will be able to change skeletal animations of 3D models. The editor will use all the other parts and modules and it will present their functionality and capabilities.

An editor for skeletal animations is a complex application with many completely different parts. It requires implementation of skeletal animation along with some extension that allows the user to change the animations or their parameters. It must be able to load and store models to the hard drive and it requires powerful graphical environment to achieve reasonable performance and picture quality. The interface must be easy to use but complex enough to offer good control over the animations. The editor will be designed to show the possible use of the other parts of this project – the implementation of skeletal animations and the message-based synchronization. The editor will present the use and extension of the other parts, and then it will contain a few other parts for completeness, like a graphical interface and a file management. The editor will represent a complete path from a conceptual animation to a real animated model that can be moved and changed.

1.9. Project goals summary

This project has two main goals: 1) complete and open implementation of skeletal animation for 3D models, using key-frames, 2) implementation and comparison of message-based synchronization in multithreaded environment of graphical applications, and an implementation of a simple editor for skeletal animations.

1.10. Minor goals

The project is aimed at skeletal animations and parallel programming, but there are some

other parts that must be implemented in order to make the editor complete. These parts are present, but they are not so developed and they are not discussed in such a great detail. Among them is: Working with the model files – parsing and loading of the models is left to the DLL libraries and the basic implementation supports only a few formats or their versions. Changing the model geometry and skeleton – work with the geometry ans skeleton is very difficult and to allow the user to change them would add additional complexity to the editor even though it is not the main target of the project.

2. Analysis

This chapter presents solutions to the most important problems encountered during the implementation of skeletal animations and then it discusses the Dragonfly animation editor. Multiple solutions are presented to every problem and then the best is selected, based on its efficiency and overall properties.

This chapter is divided in two parts, the first part discusses the problems of skeletal animations and the second part presents solution to problems of a graphical framework and the Dragonfly editor which is based on that framework.

2.1. Skeletal animation

2.1.1. Overview

The skeletal animation is an effective and flexible technique but it is also hard to implement. This section explains the mechanics in a greater detail and it addresses the most important problems encountered during the implementation. Each problem or important part is discussed separately and a solution is presented along with some other possible approaches.

2.1.2. Data representation

Effective data representation is very important because the skeletal animation has very specific requirements on the data format and organization. The model consists of three main parts: 1) geometry with a skinning information[8], 2) skeleton, 3) animation data.

2.1.2.1. Geometry representation

The geometry representation is very similar to any other model format: the surface is constructed from triangles and each triangle is constructed from three vertices with skinning attached. The skinning information is used to link bones and vertices and each vertex must contain complete information about all the bones that influence it. The most common method is that the vertex contains a list of bones and a list of coefficients determining how much is each bone significant. This way a vertex can be influenced by any bone and the model will be very compact and flexible. There are other feasible solutions:

- Each vertex is influenced by a single bone, this way, some space is saved and a few of matrix multiplications are removed, but the animations must be very simple or else the the data will not be able to represent it.
- Other possibility is that there can be only several bones loaded in the video memory and each vertex contains the coefficient for every bone. This method is very wasteful because the model must be either very simple or the bones must be frequently changed and the implementation is not very flexible.

Therefore, the first solution is the most flexible and it is effective enough that it is used in almost exclusively. The only exception are systems that do not support programmable shaders. The vertex contains a list of four indices and four coefficients and there are only four spaces because tests showed that any bigger number does not bring any significant improvement in the resulting

8 The binding between the skeleton and the vertices.

animation, according to [Möller02]. Then each vertex contains a position vector, a normal vector and two texture coordinates, for two textures. These coordinates can be used to map textures, bump maps or normal maps and their actual meaning depends on the effect used to render the model. The vertices are stored in a vertex buffer that is placed in the managed memory pool[9]. They are used to construct triangles and they are organized via an index buffer. Thus the geometry is stored in two separate buffers and it is loaded in the video memory all the time, unless the graphical device becomes lost[10].

2.1.2.2. Skeleton representation

The skeleton is a hierarchy of bones, that is shaped like a tree (graph). The skeleton must contain an information about the relationship between bones and their binding poses – offset in the absolute coordinate system. Each bone must have the transformation relative to its parent, but this is the part that is changed during the animation and therefore it is stored in the animation data. The offset is a simple transformation from the absolute space to the bone space and it is never modified, therefore it can be stored as a simple matrix. In the hierarchy each bone can have many children, but only one parent and that is why each bone has only information about its parent with the root bone having an empty reference. The reference can be a pointer, but it is better to use an index (number), because it can be stored in a file without any changes or conversions and if the skeleton is used by shaders then it must be a number, because shaders can not use pointers.

2.1.2.3. Animation representation

The animation data is the most complicated data structure in the model. It must contain the data of multiple different animations, called animation takes. Each animation take must store a transformation for each bone in every key-frame. An animation take can store its key-frames in two ways: The take can contain a list of key-frames, each with one transformation for each bone, which is a simple representation that has no big advantages or disadvantages. Or the take can store the transformations in a separate list for each bone, then each bone has its own list, called track, of key-frames, each with a single transformation. This method takes a little more space, but it allows some transformations to be made on the model data. For example – the animation can be easily compressed by omitting the same key-frames. That can be done with the previous representation too, but the entire key-frames are rarely identical, the bones on the other hand can have the same transformation in many key-frames and some parts don't move at all, like the legs when the model is just waving hands. The second representation is more flexible and it makes the model more extensible.

2.1.2.4. Data serialization

The data hierarchy presented in the previous sections is used while a model is in a memory and when it is used or rendered. When a model needs to be stored into a file or it has to be sent somewhere else, then it may be a good idea to serialize the model to a simpler representation, like an array of bytes. There are many ways how to do that, but the most common is a binary serialization – the model is transformed into an array of bytes. All the data parts are just transformed into bytes and then they are stored in a single buffer.

---

9 The managed memory pool is controlled by the DirectX run-time and such resources are placed in the video memory if it is possible. The run-time is responsible for their transfer.

10 Direct3D graphical device ca become lost as a result of a few important actions – graphical driver failure, the application using the device is minimized or suspended or some other application requests the graphical interface. When a device is lost than it can not be used to any rendering and the application must wait until it can be reset. More information can be found in [MSDN4].

2.1.3. Skeleton implementation

A skeleton is a very important part of a model, because it defines models shape and movement. It is mainly used to compute the skinning transformations and it is most efficient to compute the transformations using a graphical hardware, because the hardware is very fast and it is designed to perform such computations effectively. The skeleton must contain an information about a bone hierarchy and it must store an offset matrix for each bone. Although the skeleton does not contain that much data it is used by the other parts as a link. Mainly, vertices and animations address bones by their index in the skeleton. Even though there other ways to address bones (pointers or references), the graphical hardware is able to work only with numbers and it is good to use the same mechanism in the entire application. The skeleton can be constructed like a hierarchy of bones connected by pointers, but this would mean that shaders would have to use some other representation, because they can not work with pointers.

2.1.4. Skin implementation

The skinning is defined as a list of four indices and four coefficients and the final position of a vertex is computed according to this formula:

$$position_N = \sum_{i=0}^{3} skeleton[index_i] * position_O * coefficient_i$$

- position N is the final position of a vertex.
- skeleton is a list containing a final position for every bone.
- index and coefficient define the skinning of a vertex.
- position O is the original position of a vertex.

The bone transformations are all applied on a vertex and the result is modified by an appropriate coefficient, therefore the bones with a bigger influence leave more of an impact then the others. If some bone has influence of zero, then it is not used at all and it can be omitted, but that would require a conditional jump and that may hurt the performance of the graphical hardware[11]. The final position can be later modified by some other factors, like a world transformation. This computation is usually done completely by the graphical hardware.

2.1.4.1. Multiple animations

The model and its skeleton is designed to allow users to render multiple animations at once. The animations can be combined the same way as the different bones are combined in the skinning transformation. When there is used more then one animation then the final transformation is computed for each bone in every animation and the result is combined into one final matrix and that is used to move the vertices. It is possible to compute final a transformation for every animation and combine them later, during the vertex transformation, but this would add unnecessary computations.

2.1.5. Animation implementation and used computations

A final pose of a bone is required to compute a final pose of a vertex and the final pose for each bone must be computed first and it must be loaded into a video memory for maximal hardware acceleration. The bone transformations are usually computed on the CPU because the bone hierarchy is small and the computations must be done before rendering. The hierarchy size is limited by capabilities of the hardware and it is explained later in the section discussing shader[12]

11 Graphical hardware is designed for extremely fast computations without any jumps or indirect access to variables.

12 A simple program used to control the work of a graphical hardware.

design.

2.1.5.1. Bone transformations in a key-frame

First, it is necessary to discuss how to compute the final transformation of a bone in a key-frame. A bone is influenced by the transformations of its ancestors and its own offset. The to-root transformation is computed recursively for each bone from its transformation and the transformations of its ancestors. Every change in the bone pose affects the position of all its descendants and this is exactly what the skeleton should do – when an arm moves then the forearm and the hand should move with it. The formula is:

$$ToRoot_i = ToParent_i * ToRoot_p$$

- ToRoot i is the complete relative transformation of a bone – its relation to the center of a skeleton.
- ToParent i is the main transformation of the bone – relation to its parent (it is stored as part of the actual animation).
- ToRoot p is the relative transformation of a parent bone.

The recursion must begin with the root bone and it must continue for each child. The ToRoot transformation is called the relative pose because it is relative to the skeleton but it is not valid in the absolute space of the entire scene. It can be useful for some special effects, for example: it is used to render the models skeleton in the animation editor.

The final bone transformation is computed from the relative pose according to the formula:

$$Final_i = Offset_i * ToRoot_i$$

- Final i is the final transformation of a bone used to move models vertices
- Offset i is the binding pose of a bone – its offset from the origin of the entire scene.
- ToRoot i is the relative transformation of a bone – its relation to the center of a skeleton.

This is the final transformation of the bone that can be used to move the vertices and it is applied on them according to their skinning information.

2.1.5.2. Bone transformations between key-frames

The formulas presented in the previous section are used to compute bone poses in a key-frame, but bone transformations are usually computed between key-frames and to do that, it is necessary to compute an appropriate transformations from the key-frames. It is possible to use the closest key-frame or many key-frames can be combined to create a more smooth movement and animation. Any technique is valid as long as the resulting animation meets the expectations. There are three simple methods that can be used as an example:

1. A linear interpolation.
2. Select and use the the closest key-frame.
3. Select the last, preceding key-frame.

The adjacent key-frames are selected based on the current time and their time info. The second and the third methods are pretty simple, but the linear interpolation is a little more complex. First, two key-frames are selected, one just before the current time and one directly after the time, and then these key-frames are combined based on their distance from the current time. For example: The animation is 1 second long with a key-frame set for every 100 ms, if the time is 350 ms then the

final pose is computed from the 4th and 5th key-frame. The two key-frames are combined simply 0.5 * K1 + 0.5 * K2, because the the time is just half way between them.

When key-frames are combined then the result is used to compute the final transformations and the easiest way is to create a temporary key-frame and use the exact formulas presented in the previous section. It is possible to use a different method, but it adds additional complexity to the solution even though it is technically the same computation.

2.1.5.3. Computation complexity

After a final transformation is computed for each bone then it is used to move vertices using the formula explained earlier. Bones are prepared and then combined together for each vertex according to its skinning information and the final position is used to render the vertex. This is the main idea of the skeletal animation. To achieve this it is necessary to prepare all the required parts and then they must be combined together using the CPU and the GPU. The complexity of such computation is $2*O(N_B)+4*O(N_V)$, where N(B) is number of bones, N(V) is number of vertices and the basic operation is a matrix multiplication. The first part represents the computation of the final poses for bones. The first multiplication computes relative transformations and even thought it is recursive, it can be done in the time O(N) using the DFS[13]. The second multiplication is the offset transformation. The second part represents the skinning transformation that uses four bones for each vertex, which means four multiplications per vertex. The first part is done by CPU and the second part is done by GPU.

When a skeletal animation is applied on each vertex then the result is a new 3D mesh that is derived from the original one. To render it properly, with all textures and shading, it is necessary to prepare all the required resources, they must be loaded in the video memory and then they can be applied to the vertices.

2.1.6. vertex properties and materials

Each vertex has many specific parameters that are required to compute its color and position. There are some properties that are different for each vertex, like a position and a normal vector, but then there are properties that are the same for many vertices, like a texture for an entire door. A material[14] can be used to set some of the properties very effectively. It is a package containing all the values that is loaded into the video memory as a single object and it is used to set properties that are common to some bigger group of vertices. The properties could be loaded separately, but that may be considerably slower. All the materials are loaded in the video memory (if it is possible) and each vertex has an index identifying its material. Therefore, the vertex contains it unique information, like its position, normal vector or texture coordinates, and then it has a reference to the material list and all this information is combined during the computations to produce the final position and color. A vertex can contain all the required information, but that would considerably increase the size of a model which would lead to unnecessary redundant data.

2.1.6.1. Materials

The material usually contains two main objects: 1) lighting and shading information and 2) texture information. It can have many other parts, but those two are the most common.

---

13 Depth-first search algorithm for tree graph processing.

14 Material is a set of properties common to a bigger number of vertices that is used to set these properties for all the vertices with a single call.

2.1.6.2. Lighting parameters

A lighting or shading information represents the ability of an object to absorb or reflect light and there are many lighting models that can be used to compute lights intensity and its color. Each model requires the vertices to have different properties because the computations typically differ. The most commonly used model is the Phong lighting model, because it is very efficient and it offers very good picture quality. To work correctly with this lighting model, a material must contain an ambient color, a diffusive color, a specular color and a specular power. Other lighting models can be used simply by changing the material format and the shader implementation according to the selected model.

2.1.6.3. Textures

A texture is an image rendered on a surface of a model according to texture coordinates specified by vertices. The coordinates are stored with the vertices because they are unique for every vertex. The material is responsible only for the management of the texture data – a simple image. Therefore, the material must contain the image data and its parameters, like its width and height. If the texture is not loaded and stored with the model, it is a stand-alone file, then it might be a good idea to store its file name, so it can be loaded and stored separately.

2.1.7. Extensions and additional functionality

If a model has a geometry, an animation data, a skeleton and a material list then it is complete and it can be rendered correctly. But sometimes, the model must contain some other information or functionality so it can be used for more then just rendering and there are applications that require many complex extensions to be able to use the model correctly. Among such applications are video games, simulations or some forms of animation (computer animated effects in films).

2.1.7.1. Basic functionality

Basic functionality contains the animation computations and rendering procedures. Then there is a functionality that allows computation of relative transformations and a simple bounding volume. The relative transformations are used in some specific situations, like the rendering of a model skeleton. The bounding volume can be a simple sphere or a box around an entire model, that is used for correct placement of the model in a scene or it can be used for hit detection.

2.1.7.2. Additional functionality

Many applications, like video games, use skeletal animations, but they require various additional functionality besides the basic animation and this section presents the most common extensions that might be required by these applications. The extensions should use the basic skeletal animations to animate the model and then the extensions should offer various additional functions that are not related to the animation, but they use the same data and basic functions.

For video games (and ray-tracing) is very important a hit detection. The hit detection can be done simply by testing every triangle of the mesh, but that is very inefficient and any real-time application (like a game) must use some faster technique. These techniques usually use hierarchy of bounding volumes for each part of the model organized like a tree or some other topology structure. The basic bounding volume can be used for the most general test, but each application must implement its own functionality for more accurate tests. This would require additional functions for creation and storage of the hierarchy.

Another very complex extension might be a progressive mesh. The progressive mesh is a 3D

mesh that can be transformed to more or less complex version of the original. This method is used for complex models when they are far from the virtual camera. In that case the model is transformed to a more simple version and then it is rendered making the computations much easier. This method can be very effective, but it is very difficult to implement in combination with the skeletal animation.

Video games might also require the option to change the animation during the presentation because that allows them to precisely reflect some user actions, like if a player avatar is running the same time it is aiming its gun. Such situations can be solved by using multiple animation, but the avatar can turn in many ways and it is much more effective if such movement is achieved via a simple transformation that is combined with the actual animation. This is a simple extension that can be implemented by using a list of additional transformations for every bone.

There is an extension that is useful for many different applications – data compression. Compression can be applied on a geometry, a skeleton or animations. Compression of the geometry is the same as for the standard 3D mesh and the skeleton is very simple and small and its compression does not bring any significant improvement in the resulting size. The animations are much bigger and there can be achieved a big reduction in the model size. There are many ways the animation can be compressed, but the most common method is an elimination of redundant data. The data representation, presented in the previous sections, is designed to allow a basic compression and it is primarily aimed at this method.

2.2. Graphical framework and animation editor

This chapter presents solutions to the most important problems of design of a graphical framework and an animation editor. This chapter presents the animation editor as a whole and it does not separate the editor and the graphical framework, because the main goal is to implement an animation editor and the graphical framework is just its part. These two part are discussed together, because they are closely related. The framework is not constructed as a universal graphical library, instead it is a simple graphical environment designed for the animation editor and that is the main reason why it is not discussed separately. In the user manual, there is a programming tutorial that shows how the framework can be used separately to construct a window and graphical environment to render animated models.

2.2.1.1. Animation editor overview

A graphical animation editor is an application that allows user to load animated models, they can change models animations in a graphical environment and then they can save the models back to a file. The editor is a window application with graphical user interface (GUI) and it is constructed from many different parts, like the user interface, graphical environment for model presentation or modules for loading and storing of the models. Therefore, the editor is a complex application and every part must be carefully prepared and implemented. there are many problems that must be resolved before the editor can be completely implemented: 1) overall environment and class design, 2) model representation that allows editing, 3) threading and parallelism, 4) 3D scene management, 5) input precessing and user interfaces, 6) error handling and reporting.

2.2.2. Extension of the animated model

The basic skeletal animation implementation as it is presented in the previous chapter does not allow any changes to be made to its internal data and such functionality must be added as an extension to the basic implementation. This extension is called a model adapter and it is a wrapper for the basic model that implements the required functionality. The adapter is used directly by the

editor and its methods cover all the available operations offered by the animation editor.

To change the animations, the adapter must have access to the original animation data, loaded with the model, and it must provide a safe way to change them. There are many ways how these changes can be applied and how they may be rendered. They can be made directly to the model or they can be stored in the separately and then transferred to the model.

2.2.2.1. Animation changes

It is possible to apply user changes directly to the model, but that makes it very hard to keep track of such changes. A better solution is to store the changes in the adapter, in an internal animation data, these changes can be then combined with the original animations and the result can be sent to the model. This approach takes a lot of space, but it allows the adapter to monitor all the user changes and present them as numbers for more precise work. This is necessary, when an editor allows users to change the animations using exact values as well as some graphical interactive controls (sliders, arrows). Users can either use the graphical controls to move a bone in a position they like, or they can set the value as a precise number.

The adapter loads the animated model, then it obtains a copy of the models original animation data and it creates two internal copies of the animations. It stores the original data and then it has an empty copy, that has the same data structure (number of animation takes, etc.) but all the transformations are empty. The first copy is used as the basic position and the second copy represents the changes made by the user, then these two copies are combined together to produce the final animation that is sent to the model and it is rendered to the scene. The copy of the original data is never changed.

2.2.2.2. Available functionality

Skeletal animations can be edited by changing the transformation of bones in their key-frames. Simple Windows controls can be used to allow users to select animations and key-frames, but the bones can be rendered in the scene with the model and users should be able to select them simply using a mouse. This way, users can select bones and they can see the entire skeleton, its organization an position. For simple models, it may not be necessary, but it is very important for more complex models and it is easy to use.

2.2.2.3. Visual skeleton

To allow users to select bones a skeleton must be rendered to the scene along with the model. The skeleton must move with the model and each graphical bone must mimic the movement of the real bone that transforms the model. Bones can be constructed from a few vertices and the vertices can have a simple skinning info: every vertex belongs to a bone and it has the indices $(Index_{Bone},0,0,0)$ and the coefficients $(1,0,0,0)$. This way a vertex is only affected by its bone and no other. The bones are animated using relative transformations computed by the animated model, because that is the bone pose without its offset. The skeleton can be constructed and render many other ways, but this representation is simple and effective – the bones are simple and they are rendered all at once.

Rendered skeleton offers the visual representation of the actual bones, but it also allows users to select bones so they can be moved via other controls of the editor. The bones can be chosen by a selecting ray, that has an origin in the absolute space origin and a direction based on the mouse position:

$$Origin=(0,0,0) \quad Direction=(X_F, Y_F, 1)$$

$$X_F = (2 * X_O / Width - 1) / Projection[0,0]$$

$$Y_F = (-2 * Y_O / Height + 1) / Projection[1,1]$$,

- X(O) and Y(O) represent the real position of the mouse.
- Width and Height define the size of the scene.
- Projection is the projection matrix.

Every triangle of the skeleton is tested for intersection with this ray and the closest hit bone is returned as selected. Bones can be selected in the rendered position, because the intersection is tested on the skeleton transformed by the actual animation. This method is not so effective, but the bones are very simple and the skeleton can contain only a very limited number of bone[15]. Advanced topology can be used for optimization, but the skeleton is so simple and small that it is not necessary.

2.2.2.4. Changes in the animation hierarchy

Changes to the animations can be done by moving the bones in key-frames, but the editor should allow the creation and destruction of sole key-frames and entire animations. Both the key-frames and the animations can be created as a copy of the already existing ones and the editor should ensure that the model has always at least one animation with at least one key-frame. The key-frames can be created simply by selecting a time in an animation. The new key-frame is constructed as an exact copy of its predecessor.

New key-frames and animations must be created in the original animation data stored in the model adapter and this is the only time this data is changed. The same changes must be applied to the data containing the user changes (in the adapter) and the data of the actual model must be changed as well. This way all the data is synchronized and the work may continue without crashes.

2.2.3. Parallel environment and synchronization

Graphical applications are typically constructed from two main parts: a graphical engine and a user interface. This design is well suited for parallelism and the separation into two threads allows the application to use more processors or logical threads, if they are available. Moreover, if the graphical part is not in the same thread as the user input then the application can communicate with the user even when the graphical thread is busy of if it has some serious problems.

2.2.3.1. Message-based synchronization

This project presents the message-based synchronization method for the multithreaded computation. It is a special technique, that does not use lock for synchronization, instead the objects communicate by short messages sent between independent threads. This way, the thread simply sends a message any time it needs any synchronization and then it resumes its work. These messages are stored in the target thread for a while and then they are all resolved at once. This method has the great advantage, that threads do not need to wait for any locks and they can do other things. It has also disadvantages, while the worst is that the object can not instantly obtain response to its message, instead it must wait for it and if it needs the response for its work, then the thread will be locked regardless of the synchronization method. The great advantage of graphical applications is that their objects usually do not need response, because there is a very specific relationship between objects: the user interface accepts input from users, then it chooses the

15 This number is a based on the restrictions of the shader version 2.0 and a skeleton typically contains less then 20 bones.

appropriate actions and it informs the graphical objects, no response is required. This makes the message-based synchronization very effective for the graphical applications.

2.2.3.2. Basic characteristics

The message-based synchronization allows the threads to run without waiting for locks, but it does not support any immediate responses to the messages. The lock-based synchronization provides complete access to the shared memory for all threads, but it leads to waiting in critical areas. The locks are the most commonly used synchronization method and they are the best choice for most applications, but messages can be very effective in some specific applications, like video games. The message-based synchronization may be implemented in many different ways and the one presented here is not the only possible construction.

2.2.3.3. Thread representation

Each thread is created as new task with a main function that is specified during its creation. The presented solution uses a concept of workers and receivers to make the threads more flexible. All threads are created with the same main function that contains a simple loop which runs from the creation of the thread until its destruction and it is used to work with receivers and workers. A worker is an object that implements some functionality that needs to be run in the thread. Workers are registered in a thread and the thread gives them a time to run once every iteration of its main loop. A receiver is an object designed to process messages sent to the thread and it is registered in the thread like a worker, except in a different register. The receivers are used as targets for messages sent to the thread, where they are registered, and they serve as an interface between the thread and the rest of the application. The thread dispatches all stored messages once in every iteration of its main loop and these messages are delivered to the registered receivers. Threads resolves a few simple messages used to work with the threads themselves and with their registers, like registration of a worker. Both the workers and the receivers can be removed at any time via a simple message sent to their thread. This construction allows users to add any required tasks to the thread (as workers), register objects responsible for communication with other threads (receivers) and remove them when they are no longer needed.

2.2.3.4. Messages

A message is the main communication element used for synchronization and the exchange of data. There are two main ways it can be implemented: 1) a message is a simple object that is the same for every communication or 2) every communication uses an unique message that is derived from some simple basic message. The first version is simple and small, but it may not suffice some of the more complicated communications and they must send messages with pointers to the actual data or some other nontrivial constructs. The second implementation is bigger and more complex, but it can be extended according to the actual needs of every communication. The presented solution uses the second method, because it is more flexible.

2.2.3.5. Class design

The message-based synchronization requires very specific class design that can lead to much bigger classes, because they must be designed to work with messages and threads. If a class should be used directly with the threads, then it must be designed as either a worker or a receiver and that means that the class must implement at least some basic methods. A class should implement some methods for communication between threads to hide the messages, because that way other objects can work with it simply, like they do with other objects in the same thread. Another important rule is that the actual work should be done by the workers during their time in the threads main loop and

the receivers should execute only simple actions in response to the messages.

2.2.4. User interface and input management

As every graphical application, the editor should use a graphical interface for communication with its users. The interface is basically divided in two main parts: 1) 3D interface including model presentation and 2) standard 2D interface based on buttons and other common controls supported by MS Windows. Every part can process its input differently and it can use completely different controls (buttons, text boxes).

The entire input should be processed in a single thread, that is completely independent, because this way it may be operational even when the others encounter some difficulties or errors. The input can be processed separately for the 3D scene and for the Windows controls placed in the other parts of the main window of the editor. It can be processed differently and it can be even captured[16] differently, because the scene can use Direct Input[17] and the Windows controls must the standard window procedures[18] to receive the input. This may bring some difficulties, but it allows the application to manage every input based on its origin and meaning.

2.2.5. Direct3D environment and graphical device

The editor uses the DirectX 9 technology for the hardware acceleration, because it is stable and offers all the required functionality. The skeletal animation presented in previous sections is currently implemented under Direct3D 9, but it can be simply modified for work under the Direct3D 10 or 11, because it does not use any obsolete features, like fixed rendering pipeline.

In the environment of the DirectX, the 3D scene is represented by the Direct3D Device, that is used as a abstract model of the actual hardware adapter and it manages the rendering of all objects in the scene. The native device is very hard to maintain, because it requires a lot of parameters and every operation must be checked for safety and that is why the editor should use a wrapper that controls the actual device which simplifies the use of the device. It is very important that the initialization, the destruction and every use of the device is carefully controlled, because any unsafe operation can cause immediate crash of the entire application, due to hardware failure. The wrapper ensures that all the operations are safe and it checks the device for any special situations. It must make sure that the device is operational before any rendering, because if it is not, then it must not be used for any graphical output, instead it must be reset and all resources must be loaded into the video memory again. The wrapper must inform other objects whenever the device becomes lost, that way they can react and reload any graphical resources to avoid errors. This design is very effective and safe, and it is also very simple and easy to use. The editor could use some complex graphical framework, but this method allows it to present the skeletal animations without any hidden functionality and it is very simple.

2.2.5.1. Rendering

The graphical run-time of animation editor is designed for a simple rendering process – the application contains the main loop and the scene is rendered once in every iteration. The rendering should be done all in one place and everything should be rendered at the same time, because the

16 Users interact with applications via some input devices, like a mouse or a keyboard, and the application must obtain the input from these devices, this process is called: input capture. This method is common for video game consoles and some simple computers (embedded systems).

17 A technology designed by Microsoft as the input method for the DirectX framework.

18 Windows must have assigned a procedure that is responsible for input processing. The input is captured and dispatched by the OS.

scene[19] must be opened and presented only once in every frame. It is very important that all the rendering is done between the creation and presentation of the scene, because if something is rendered too soon or too late then it will not be rendered at all and it may cause a fatal error. The editor can maintain the correct order using threads and their workers. All the workers are called once every iteration of the threads loop and they are all called in the same order they were registered. The object responsible for scene creation is registered first and the one that handles the presentation is last and all the workers between them can safely render everything they need. There are other ways the order could be maintained, but this method shows the capabilities of the used threading mechanism and it is easily extensible. For example, the rendering could be concentrated in a single object and all the other objects would be registered there.

The graphical environment should not be used by more then one thread, it should be created in a thread, then it should be used in that thread and it should be destroyed in the same thread. Even though the current version of the DirectX run-time is able to work with more threads, it can lead to problems that are very difficult to find.

2.2.6. Environment construction and maintenance

Every application starts with a single thread that runs the main function and every other thread must be created later with its own functionality and purpose. The first thread is the main thread and it is the entry point of the entire application. For multithreaded applications is very important that the other threads are created in a precise order and that their content can be safely initialized. This is even more important for graphical multithreaded applications, because the graphical device must be created and used in a single thread, but some objects from the other threads may need the device for their initialization. Therefore, the device must be created in a selected thread and other threads must wait for it.

2.2.6.1. Application initialization

Another problem of multithreaded applications is the user input, because the input can be processed only when all the required objects are functional, but the application should respond as soon as it is possible. First, the editor must create all the objects necessary for input management and then it have to create objects responsible for input processing. This way the input is always processed correctly and the editor starts responding very soon, because it does not construct any complicated objects at the start. The model and some other graphical objects are created later, when the application is already fully operational.

2.2.6.2. Run-time

During the run-time, it is very important that the objects communicate correctly inside threads and between them. The objects that are used in the same thread should communicate using standard method calls, because it is efficient and the thread is not slowed by any unnecessary messages. The communication inside a thread is always serialized and therefore, it is always safe to use standard methods. The objects that are used in different threads should always use messages for communication, because direct access could cause synchronization problems – race condition[20]. Objects that are used in multiple threads should implement their method in two versions: the first version should be the actual method with all the functionality and the second version should just send a message to the object. This way the objects in the same thread can use the first version and

19 The scene is an image rendered by the device to its space on the screen.

20 A problem caused when two thread are accessing the same part of the memory without synchronization. The memory could be in an undefined state due to parallelism and such operation could lead to and undefined state of the application or fatal error.

any communication between the threads can be realized via the second version. A object should be owned by only one thread and it should execute its main functionality in the same thread (if it is a worker), but it can offer some services to other threads and their objects.

2.2.6.3. Application termination

It is important to maintain a precise order during the end of a multithreaded application, because the destruction can be parallel and an incorrect order could lead to a fatal error. The main concern is that objects must be destroyed after all the objects that use their functionality, otherwise some object could call a method of a destroyed object, which leads to fatal error. The graphical device is the main source of these problems for graphical applications, because the rendering must be terminated before the device is destroyed and if the device is used in multiple threads then it may be very difficult to find the correct time for the destruction. To solve this problem, the editor can simply remove all the workers from the graphical thread, because they are the only objects, that perform any rendering and then it is safe to destroy the device. This is another advantage of the centralized rendering.

The user input is a problem during the termination as it is during the start of the application, because it must not be processed after the destruction of required objects. The correct termination of the input is especially important for the graphical applications and the editor can use a simple method to do it safely: It stops accepting the input, then it destroys the objects responsible for input processing and then it can safely destroy everything else.

2.2.7. Error management and warning report

Error handling in C++ applications is a very complicated problem that usually does not have a simple solution and every application must define its own rules. There are some basic ways the errors can be handled, based on their origin and meaning:

- The programming errors, caused by a mistake of programmer, should be controlled by assert macro, because these errors should not be present in the final version of the application. The assert macro is removed in the release build and it does not slow down the resulting application.
- Minor errors, caused by an invalid user input or some incorrect file, should be reported by a simple code returned by the function where they were located. This way, the output of such functions can be simply tested and the application can react by prompting the user, for example.
- Serious or fatal errors should be reported using the exceptions. This way, the error can not be ignored and if the exception is not correctly handled then the application is terminated and it may not damage other parts of the system, like incorrect changes to a hard drive or the OS.

Another problem is that C++ does not have binary standard for exceptions, which means that an exception, created in a function from a static or dynamic library, can cause fatal error, because it may not be compatible with the try-catch blocks of the application. This may be caused by different compilers or different options used to compile the libraries. Therefore, it may be a good idea to use exceptions only inside independent modules and their interface should report the errors via return codes.

2.2.7.1. Error management used in the project

In this project, errors are handled according to the rules specified in the previous section.

Parts of the project must be added to other programs as source files, to avoid the problems with exceptions and external libraries. This may be inconvenient, because the program may need many additional files and it could become very big, but it prevents any problems caused by exceptions and it allows the compiler to use better optimizations on the included code. The only exception is the module implementing the skeletal animations, because it is designed as a stand alone library and it can be linked as a static library (the compiled library is part of the project distribution). The number of additional files is not a big problem when a project uses only the module with skeletal animations, because it is designed as a separate part and it requires only seven files (and two of them are simple headers that just include multiple files from DirectX or standard library).

Exceptions are used to report very serious errors, like problems with the graphical adapter, and the editor uses a single try-catch block in the main function to catch any exceptions that are not properly handled in the program. It prevents any sudden, unexplained crashes in case some subsystem fails or a resource is not accessible.

The editor handles differently errors caused by user input and errors caused by any internal failure. The errors during the users work are not reported, errors like invalid number input, multiple key-frames creation for one time, attempt to open non existing file. These errors are ignored and user must try again with a valid input, otherwise the editor would be really annoying, because it will report an error every time the user makes any typing error.

3. Implementation

This chapter discusses the actual implementation of techniques presented in the previous chapter and it is divided into two main parts:

The first part discusses an implement of the skeletal animations and it specifies all the classes and files necessary to use the implementation in other projects. Together, these classes can be used as an independent animation library and the first part discusses their functionality and construction. A special tutorial is included in the user documentation, this tutorial presents the animation library as stand-alone module and it explains every step, that is necessary to use the library in a graphical application, along with the DirectX library.

The second part describes the Dragonfly animation editor and the graphical framework used to render animated models. The graphical framework is not discussed separately, because it is designed specially for the editor and it is just a simple rendering system that does not offer any special functionality, which would require its own separate description. However, there is a tutorial included in the user documentation that describes, how the framework can be used separately, but alone, it offers only a limited functionality. Therefore, the editor is discussed as an entire application. Its overall design, main modules and classes are described in the second part that contains detailed information about every module, like its functionality, main classes and its relation to the other others.

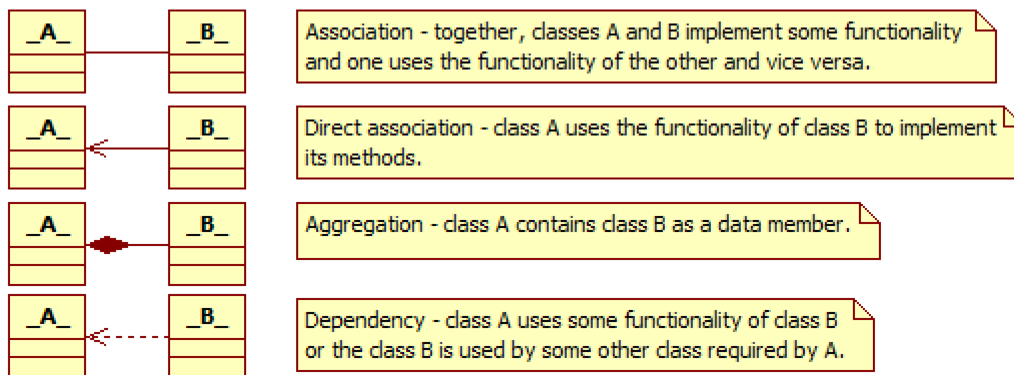

Figure 4: Description of symbols used in the diagrams to represent a relationship between classes. The only exception is the dashed line that is used to connect the notes with their targets.

3.1. Skeletal animations

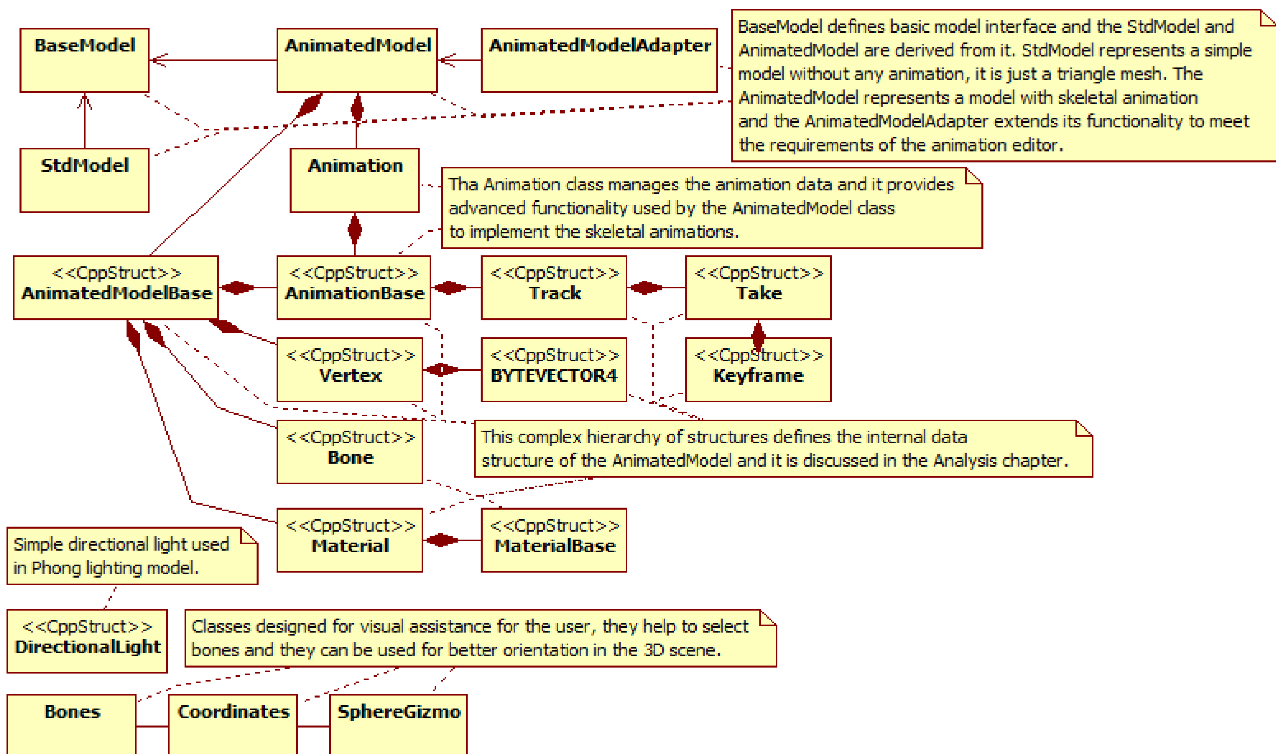

Figure 5: Dependencies between classes in the Model module. This module contains all the classes and files necessary to use the skeletal animations as an independent library. The module contains classes that are not part of the animation library, but they are closely related to the library and they work with the skeletal animations too.

The first goal of this project is to implement skeletal animations and the implementation is concentrated in the Model module. The module contains some other classes, but this section discusses only the basic skeletal animations and all the other classes are discussed in the last section of this chapter.

The Model module contains a definition of classes, that implement skeletal animations, and all the other data types required for correct animation and rendering. Together these classes form an animation library discussed in the following section. Classes implement the functionality discussed in previous chapter exactly as it is described there and this section just explains the specifics of the actual implementation. This is the main module of the project, because it contains the most important functionality that is used by the Dragonfly animation editor to render animated models.

3.1.1. Animation library

The animation library is a part of the Model module that can be used separately. It contains classes that can store and render animated models and it is completely independent part of this project. It is designed as a simple library, which can be used by any graphical application that renders its graphics using the Direct3D technology. The library is available in two versions: the first is a standard static C++ library for MS Visual Studio and the second is a set of seven files that can be included in the project like any other C++ source files. It is more comfortable to use the binary static library, but using the source files may allow compiler to perform better optimizations and that is the main reason, why there are two versions.

A project that uses the library as a set of source files must include at least these files:

- AnimatedModel.h
- AnimatedModel.cpp
- AnimatedModelBase.h
- AnimatedModelBase.cpp
- BaseModel.h
- DirectX.h
- DirectDebug.h

The first five files contain all classes and structures necessary to render 3D models with skeletal animations. The last two files simply include basic headers of Direct3D and they do not implement any crucial functionality. All these files use standard C++ library, Windows API and DirectX functions, but they do not use any other parts of the Dragonfly project.

If a project uses the compiled static library then it must include these headers:

- AnimatedModel.h
- AnimatedModelBase.h
- BaseModel.h
- DirectX.h

The last header includes main Direct3D headers that may be included separately and therefore, the DirectX.h header is not necessary, but it can be used to include all the Direct3D headers at once. The static library is available in two builds: release and debug, and it is important to link the correct version according to the actual parameters of the compiler.

There are three main classes that implement the skeletal animations and they use a complex hierarchy of structures to store the model and its animations. The basic interface that defines two main methods of the animated models is the `BaseModel` abstract class. The `AnimatedModel` class (derived from the `BaseModel`) stores and renders the models and the `Animation` class provides functions that compute bone transformations using key-frames stored in the models animations. Models are internally stored in a hierarchy of simple structures that is described in the next section and the three main classes are discussed in the three following sections.

3.1.2. Basic structures

The animated model contains a very complex data hierarchy that must represent its geometry, its skeleton and all animations and all that is stored like a tree of simple structures that is identical to the hierarchy discussed in the section 2.1.

- `Vertex` – this is a basic structure used to define the models geometry and vertices are organized in a vertex buffer for easier manipulation. The structure defines all the available vertex properties:
    1. position vector that specifies where is the vertex placed in the space
    2. normal vector which is used by the lighting algorithm and it determines the color of the vertex
    3. two 2D texture coordinates used for texture mapping while the second coordinates may be used to apply a bump map or a normal map

4. skinning information that is used by the skeletal animation algorithm and it is implemented exactly as described in the section 2.1.4.
5. material index determines the material used to render the vertex

- `Material` – this is a structure that defines a texture and lighting attributes of every vertex. Materials are stored in a material buffer and each vertex has assigned an index that identifies its material. A material stores a complete texture, its data and file name, and then it contains lighting information. The Phong lighting model, used in this implementation, combines a diffuse light, an ambient light and a specular light to produce the final color of a vertex and the material contains a color vector for each light component. The final color is created as a combination of each light component with the corresponding material vector.
- `Bone` – this structure is used to construct the entire models skeleton. A bone contains only a reference to its parent, its own index and a binding pose. The indices are simple numbers that identify bones in the skeleton, which is a simple list of bones. The binding pose is a matrix used to transform vertices from the absolute space to the bones relative space during the skeletal animation.
- `Keyframe` – key-frame is the most simple part of an animation. It defines a bones relative position in a specific time that is relative to the parent bone. The root bone stores the offset of the entire model in the scene. A key-frame contains the time when it is used and the actual transformation. The transformation is decomposed into a translation, scale and rotation parts, which is necessary for some interpolation methods.
- `Track` – a track is a list of key-frames of a single bone and it defines the bones movement in the entire animation take. The track contains a vector of key-frames and an index of the target bone.
- `Take` – an animation take defines a single animation of a model and it contains a key-frame track for each bone in the models skeleton. Besides the bone tracks, it stores an information about the length of the entire animation and its start time.
- `AnimationBase` – this structure contains all the information required for a complete skeletal animation. The first required part is a skeleton that is defined as a simple vector of bones and its hierarchy is defined by information stored in each bone. The second part is a list of animation takes that defines all the animations stored with a model. The final position of each vertex is computed using key-frames and the hierarchy information stored in the skeleton via the algorithm described in the section 2.1.
- `AnimatedModelBase` – the complete model data is stored in this structure. It contains the geometry constructed from a vertex buffer and an index buffer and then it contains the animation data represented by the `AnimationBase` structure. Besides the data, it provides a functionality for serialization and deserialization of those data.

3.1.2.1. Model serialization

When a model is transferred to an external library or a file then it must be serialized into an array of bytes, because this way it is a simple, flat data type and it can be stored without any special conversions. The serialization is defined in the `AnimatedModelBase` and it is a standard binary serialization: the data hierarchy is flattened and then it is converted into a simple byte buffer. The data is flattened by a simple depth-first search algorithm and the data is ordered as follows:

- number of bytes of the entire serialized data (this information is a 4 byte integer and it is

computed and stored at the end of the entire serialization process)

- number of vertices
  - size of each vertex
  - serialized vertices
- number of indices
  - size of each index
  - serialized index buffer
- number of attributes in an attribute buffer (this is a buffer that contains the material reference for each vertex)
  - size of attributes
  - serialized attribute buffer
- number of actual materials
  - size of a material
  - serialized materials
- number of bones
  - size of each bone
  - serialized skeleton
- number of animation takes
  - size of a take, a track and a key-frame
    - time information of a take
    - number of tracks in the take
    - serialized tracks (they are serialized the same way – number of key-frames and their serialization)

This implementation may have problems with type conversion and size, but it is very simple and it was chosen because it is very comprehensive and any user can understand it easily. It was designed as a simple method to store models rather then an universal format for representation of complex 3D scenes.

3.1.3. BaseModel interface

The `BaseModel` is a simple interface that defines two basic methods. These methods ensure that a model can render itself and than it must be able to compute its bounding sphere. Both methods are virtual and every model must implement them.

- `Present` – this method is responsible for complete and safe rendering of the model.
- `GetBoundingSphere` – the method must compute correct bounding sphere that can be used to place the model correctly in the scene. The sphere should contain all the models vertices or at least the most important parts of the model.

3.1.4. Animation class

The `Animation` is a class built on the `AnimationBase` structure and it implements the basic functionality that is required for skeletal animations. The class uses the data from the `AnimationBase` structure to compute final pose for each bone in the skeleton and these poses are later used to animate models vertices. The final poses are computed according to the algorithm described in the section 2.1. The computed poses are stored in a special vector, that can be accessed via a method, because this way the final matrices can be easily transferred to the video memory so they can be used by programmable shaders. The three main methods are:

- `BindBonePoses` – the output vector is filled with bone bind poses.
- `ToRootBonePoses` – the method computes a relative transformation of each bone, as described in the section 2.1, and the resulting matrix is stored in the output vector.
- `FinalBonePoses` – this method performs the complete computation as it is described in the section 2.1. The final poses are stored in the output vector and they can be applied directly to the models vertices.

3.1.5. AnimatedModel class

The `AnimatedModel` class is built on the `AnimatedModelBase` structure and it provides a complete implementation of a model rendering along with the `Animation` class that manages the animations. This class provides methods for rendering of the models geometry, but it does not contain the application of the bone poses that are computed by the `Animation` class, because such process depends strongly on the used effect and rendering method. The animation application must be done by an extension that uses this class and this way it can be designed according to the needs of its user. It also provides access to the internal data structures which is not a standard design for classes, but it allows the extensions (model adapters) to provide better functionality and the `AnimatedModel` does not need to be too complex. The main idea is that the extension must be a standard, completely safe class that implements its functionality based on the `AnimatedModel` class and this way it hides the unsafe design of this class while the implementation is as simple as possible. The `AnimatedModelAdapter` class used by the Dragonfly animation editor is an example of a complex extension that implements additional functionality (it provides functions that can change models animations). Main methods are:

- `Present` – this method performs the actual rendering, it must be provided with a valid graphical effect and it simply sends the geometry to the graphical hardware using the effect. The vertices are send simply as they are and the skeletal animation must be implemented in the provided effect. The final bone poses must be prepared in the effect before this method is called, because it does not send them there.
- `GetAnimation` – this method return a reference to an instance of the Animation class. This reference can be used to compute the final bone poses that must be transferred to the effect so it can animate the models vertices.
- `GetModelData` – this method provides standard (not constant) reference to the internal data hierarchy of the model. Owner of the class can access and change directly the data and these changes are not checked for validity. This approach may be very dangerous, but it offers tremendous possibilities to the extensions build on this class. An example of this concept is the `AnimatedModelAdapter` described in the next part of this chapter.

3.2. Animation editor

This section describes implementation of the Dragonfly animation editor. The editor uses skeletal animations presented in the previous section to render animated models and it is build on a simple graphical framework. The framework is responsible for creation of windows and graphical environment, but it is discussed along with the editor, because it is designed as an integral part of the animation editor. Besides the graphical framework, the editor contains modules implementing GUI and then it uses an adapter[21] for the animated model presented in the previous section.

The editor is constructed from many separate modules that provide specific functionality and together they create an entire graphical application. There are modules of two types:

1. basic modules – they are completely independent and they implement some basic functionality, like a window management
2. complex modules – these modules depend on some other modules and they implement advanced functionality.

3.2.1. Technical specification

This project presents an animation editor that is designed for the Microsoft Windows OS (Windows XP or newer), that uses the Microsoft DirectX library for any advanced hardware management and it is called the Dragonfly editor. It is build using the DirectX 9 (August 2009 build), because it is very stable and reliable version that is by far the most widely supported. The editor is a standard Windows application with a main window and a few special dialogs for an advanced input. It uses the skeletal animation implementation presented in the previous sections with the Direct3D as the only rendering technology. It uses a complex extension for the animated model to allow the user to change it. Internally, it is a multithreaded application with two threads: one for the graphics and one for the user input and other computations.

3.2.2. Module design

Every module contains a few related classes that either implement its functionality, serve as its interface or provide services to other classes. A module contains one main namespace and all the classes (and other data types) are members of this namespace. The general classes that are used only inside the module are placed in the main namespace and the other classes have their own namespace that is inside the main namespace. This way, the classes are separated and their names can not collide with the others. For example the module `Controls` contains one main namespace `Controls`, where are placed its resources and an exception class, and this namespace contains a namespace `ControlFramework` that belongs to a class of the same name.

21 Model adapter is a class that uses the `AnimatedModel` class and it offers some additional functionality along with the basic functions provided by the `AnimatedModel`.

3.2.3. Module description

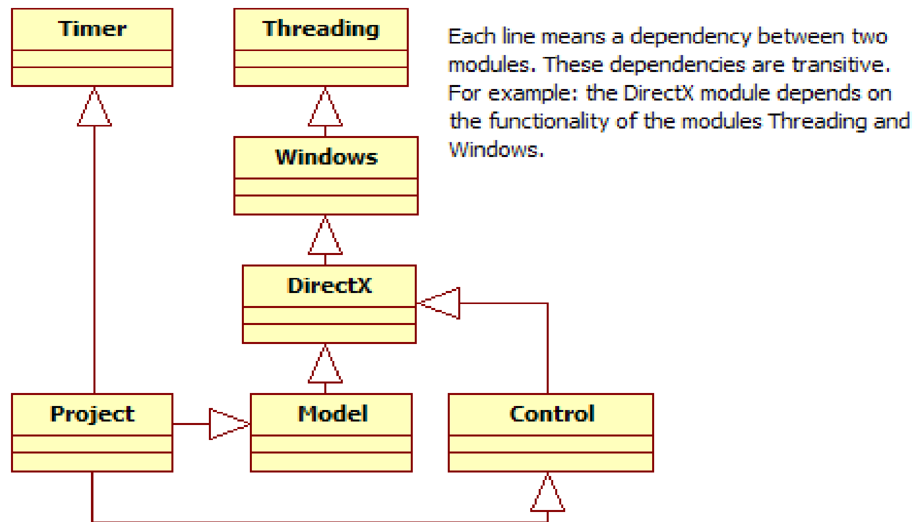

Figure 6: Dependencies between modules in the project.

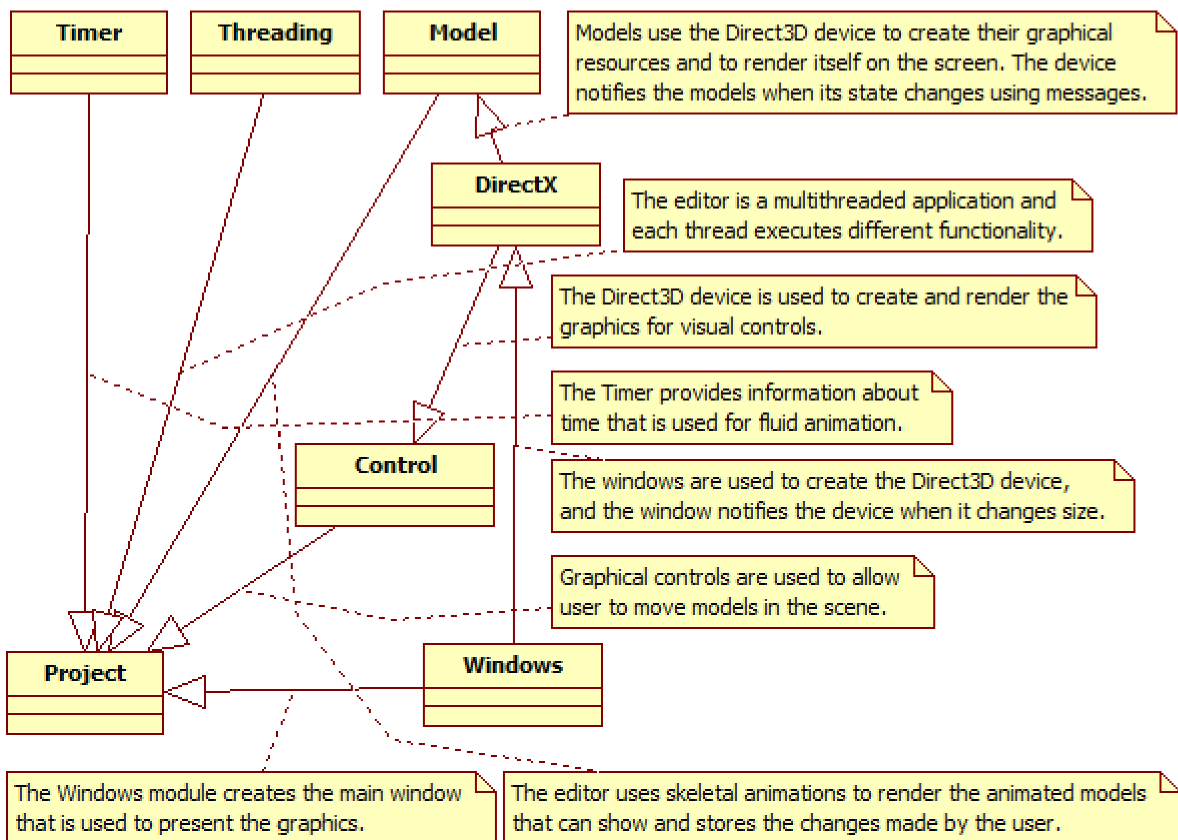

Figure 7: Basic relations between modules including the common data exchange and the most important classes.

Each module implements a specific functionality that is used by other modules (a basic

description is in the Figure 7) or it is called directly from the main function. These dependencies can not be cyclical because it may be impossible to compile such project. Every module provides one or more main classes that are used by the others and they are the only thing that is exported from a module (modules do not export stand-alone functions).

The Windows module exports the Window class that creates a single window and it stores all required parameters. This class is used to construct the main window of the animation editor.

The DirectX module contains two main parts: The first is a class representing the Direct3D device used by the rest of the application to manage the graphical resources. The second is a class that manages the Direct Input interfaces and it is used by the Project module to control user input.

The Control module implements graphical controls used to allow users to work with the animated model directly in the 3D scene. The controls are organized in a framework that uses the Direct Input for communication with users.

The Model module implements the skeletal animations and it offers some additional classes necessary for complete implementation. It provides a model adapter (extension) designed for the animation editor, because it adds the functionality required to allow users to change animated models. The classes are implemented using to the solutions presented in the section 2.1. Main part of this module is the animation library presented in the previous section. The library is not discussed in this section.

The Threading module contains classes representing threads and messages. They are both used together to implement a parallel environment with a message-based synchronization. The threads are used by the Project module to create separate threads for user input and graphical operations.

The Timer module provides a simple singleton Timer class that is used by the Project module to render fluid animations.

The Project module contains the functionality specific for the animation editor and it uses other modules to implement its classes. It creates two threads, one for user input and the other for graphical operations, then it creates the main window and Direct3D device based on it. All these objects are used to construct user interface and models with skeletal animations.

3.2.4. Run-time design

The editor is constructed from many modules, but the main module, that uses all the functionality of the others, is the `Runtime` module. It contains four main classes that are used to setup and maintain the graphical environment and input and. Users can define their own run-time class[22] that implements any additional required functionality. The custom run-time class used to implement the editor is the `EditorRuntime` and it is defined in the `Project` module, that contains a functionality specific for the animation editor. Therefore, the `Runtime` is the main module containing the general run-time classes and the `Project` module contains their specialized versions.

22 The run-time class identifies the data type which defines the main functionality of the application – it defines the run-time.

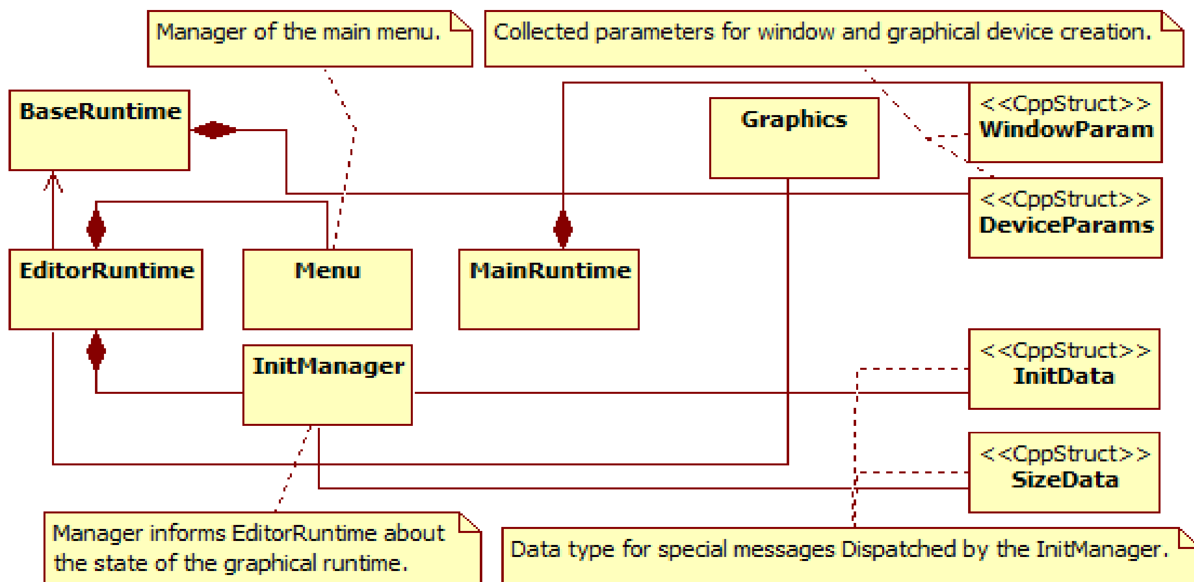

Figure 8: Dependencies between classes from the Runtime module and the Project module.

3.2.4.1. MainRuntime class

The `MainRuntime` class implements the main functionality of the editor and it is used directly by the main function to create the basic objects of the entire application. It cooperates with the `Graphics` class and it uses an implementation of the `BaseRuntime` interface to create a specific environment. It has two main method:

- `MainRuntime` – the constructor is responsible for construction of the basic objects. First, it creates the main window of the application and it captures[23] the main thread (creates a new `Thread` object, that manages the main thread). The main window is created using parameters specified by the custom run-time object (the EditorRuntime). Then, the constructor prepares the custom run-time object and creates an initialization object that ensures a proper start of the user interface[24]. Then, it creates the `Graphics` object, that manages the graphical thread, and finally, it starts the main loop of the application. The main loop is registered in the main thread as a worker and it is responsible for management of Windows messages.
- `Start` – the main function of the main thread. This function is created during the capture of the main thread and it starts the threads standard functionality, like the message processing and worker management.

The main run-time is the entry point of the application that prepares the other objects and when they are ready and running, then the run-time launches the main loop and it waits for the end of the application. At the end, the run-time destroys all the remaining objects and it terminates the application.

23 The first thread of the application is created by the OS before the application starts and it must be captured later so it can be used like the other threads.
24 Explained later.

3.2.4.2. Graphics class

The `Graphics` run-time class is responsible for initialization of the graphical environment and its management. It creates the second, graphical thread and it ensures creation and presentation of the 3D scene. The graphical thread is created in the constructor and the graphical environment is initialized later in the graphical thread, in the first time the thread calls its workers (this class is registered as a worker).

- `SetAddress` – this methods sets the callback address of the object which must be notified after initialization. This object is usually the custom run-time derived from the `BasicRuntime` interface. This address can be set using the constructor.
- `Receive` – the method used for message processing. When the application is about to be terminated, than the main run-time informs the `Graphics` run-time that it should destroy its graphical environment and the termination message is processed by this method.
- `Update` – this is the method used by workers to execute their work in the thread. The `Graphics` run-time is registered in the graphical thread as the first worker and in this method it prepares the scene for rendering, which is done by the other workers. The run-time initializes the graphical environment during the first execution of this method. It creates the device and then it informs the custom run-time when all the objects are ready and operational. The scene is presented by and instance of the `Present` class, that is owned by the `Graphics` class, and it must be registered as the last worker in the graphical thread. It is re-registered as last after a new worker is added and the `FinalizeRegistration` method is used to do that.

The `Graphics` run-time manages the graphical environment. It creates the graphical thread and registers itself as a worker, then during the first call of the `Update` method it creates the graphical device and it informs the other objects that the device is ready for rendering (so they may register their own workers responsible for the actual rendering).

3.2.4.3. BaseRuntime interface

The `BaseRuntime` is an interface for classes implementing a custom functionality used by the `MainRuntime` and the `Graphics` to create a specific environment for the application. It defines two methods that must be implemented by every custom run-time class:

- `Initialize` – this method is used by the `InitManager` when all the required objects are correctly initialized. The method receives a reference, to all the important objects, through this method and it may use the reference to create some other objects that require their specific functionality.
- `Destroy` – this method is called before the end of the application and it should destroy everything owned by the object.

This interface is designed for work with the `InitManger` and other main objects to allow the editor implement its own specific functionality.

3.2.4.4. InitManager class

The `InitManger` is a simple class that is used by the `MainRuntime` and the `Graphics` class to ensure a correct initialization and an on-time destruction of the custom run-time class derived from the `BaseRuntime` interface. It contains two main methods:

- `Initialize` – a method called by the `Graphics` object when it successfully finishes the initialization of the graphical environment. This invokes the Initialize method of the custom run-time class.
- `Destroy` – this method is used by the `MainRuntime` object to inform the custom run-time class about the program termination.

The manager is a simple class that hides the actual communication, that is message-based. This way is simplifies the custom run-time class, because it does not have to manage the entire communication. It must implement only two methods and the rest is done by the manager.

3.2.4.5. EditorRuntime class

The `EditorRuntime` is derived from the `BaseRuntime` and it is used as a custom run-time class to implement the editors functionality. It manages the user interface and it prepares all the objects that implement the interface. Most of its functionality is concentrated in the `Initialize` method, because it is derived from the `BaseRuntime`, and this initialization is called after the graphical environment is fully operational. The Initialize method is called with the most important objects as parameters – the method gets the main thread, the graphical thread, the graphical device and the main window. The main objects are used to construct the rest of the interface. These are the most important objects:

- `DirectInput` – an object that prepares and manages the Direct Input technology, it is a part of the DirectX module.
- `ControlFramework` – a framework for controls used to work in the 3D scene, it is a part of the `Control` module.
- `ModelControl` – a special control that allows the user to move the model in the scene, it is a part of the `Control` module.
- `PickControl` – a control used with the model skeleton to allow the user to pick single bones, it is a part of the `Control` module.
- control panels – classes responsible for management of the windows controls and panels, they are all part of the `Project` module.
- `AnimatedModelAdapter` – a class extending the `AnimatedModel` class, it adds the functionality necessary to change the animations (which is required by the editor), it is a part of the `Model` module.

The class offers a few new methods, except the standard ones required by the `BaseRuntime` and they are mostly responsible for work with the animated model. The two most important methods are:

- `OpenModel` – the method used to load a model from a file and the model is then rendered to the scene and the user can work with it using the editors functionality. An external DLL library is used to open and parse the model file and its serialized data is used to construct the model.
- `SaveModel` – the method stores a model into a file using an external DLL library.

The class builds entirely on the concept of the `BaseRuntime` and it is constructed to work correctly with the rest of the main run-time classes (`MainRuntime` and `Graphics`). Its main job is to create, manage and destroy other important objects, that provide the actual functionality. It

serves as an abstraction layer between the objects and the multithreaded environment, because the initialization can be complicated and everything must be created in an exact order.

3.2.4.6. Time line summary

The editor is a complex multithreaded application and its objects are created in a very specific order, that ensures their safe creation and functionality. This is the overview of the time line based on the main classes which are described in the previous sections:

- `main` function – the main function is an entry point to the application that is called by the OS loader
- `MainRuntime` – the main object, that is created in the main function and it is responsible for creation of the graphical environment and the custom run-time class based on the `EditorRuntime` interface.
- `Graphics` – this class manages the graphical environment and it creates the second thread (all graphical operations run in the second thread).
- `Graphics` Initialization – the initialization of the graphical environment is called from the second thread and when it is finished, then the rest of the application can be constructed. The initialization is not part of the construction of the `Graphics` class, because it is done in a different thread.
- `EditorRuntime` – custom run-time class derived from the `BaseRuntime` that manages most of the editors functionality. It is created along with the `MainRuntime`, but it is inactive until the Graphics class completes its initialization.
- `EditorRuntime` Initialization – the initialization stars after the `InitManager` receives a message from the Graphics initialized, that everything is properly initialized. It is not part of the construction, because if must wait for the second thread.
- **run-time** – the application is running and all its functionality is managed by the objects created during the initialization.
- `EditorRuntime` destruction – the destruction is called after the `InitManager` receives a message, that the application is about to be terminated. All the objects created by the custom run-time class are destroyed during this process. This must be done before the graphical environment is destroyed and it must release all the allocated resources.
- `Graphics` destruction – the graphical environment is destroyed with all objects used to render the 3D scene (graphical device, vertex description, custom shaders and effect).
- `MainRuntime` destruction – the main run-time object terminates the main loop and it releases all the remaining objects.
- `main` function end – the main function ends after the `MainRuntime` releases all its objects.

This is a simple overview of the editors start, its run-time and its end. The order of these actions is very important, because every object requires some other objects for it to function and they must be ready on-time.

3.2.5. Timer module

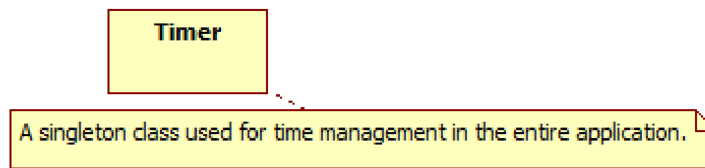

Figure 9: Dependencies between classes in the Timer module.

The `Timer` module is very simple and it is different from the others, because it does not contain its own exception or resources and the only class in this module is the `Timer`. The `Timer` class offers a very precise measurement of time and it is designed as a wrapper for a few Windows API functions.

3.2.5.1. Timer class

The Timer class is designed as a singleton so it could be used by the entire application and it can not static, because it must store a few information, that are not known during the compilation time. It uses the frequency counter to measure the time because it is very precise and it is a part of the Windows API which means that no additional libraries are required.

`GetStartTime` – this method is used to obtain the time when the timer was created.

`GetTime` – get current time.

`GetElapsedTime` – compute the time that elapsed from the time passed as a parameter. The previous time stamp must be provided by the caller and the `Timer` does not store it. This design allows the timer to be used by many objects without problems.

3.2.6. Threading module

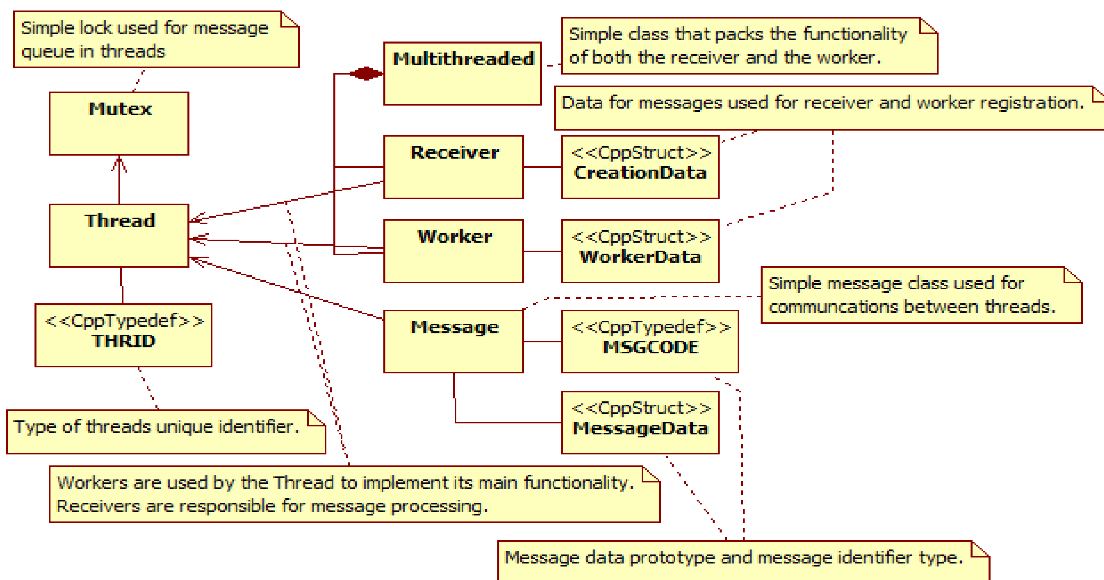

Figure 10: Dependencies between classes in the Threading module.

The Threading module is a standard module that defines its own exception type and it has its own resource class. The module contains classes implementing the message-based synchronization method, and then it defines many interfaces for the other modules.

3.2.6.1. Thread class

`Thread` is the main class of the module, because it implements the main functionality, and all other classes work with it. Each instance of this class represents a single thread, that is created during the construction, and it is started with a simple main function. The main function is similar to a main function of some windows applications, because it has a single main loop and each iteration is divided into two main parts: 1) the thread locks its message queue and resolves all messages sent to it since the last iteration and 2) the thread calls all the workers and they can do whatever they need. The queue could be implemented using a lock-free algorithm, but this way the thread could not be overloaded by messages, because it just locks the queue, then it quickly delivers every message and then it can continue with its work. The messages are delivered to receivers, based on their identifier in the threads register and some messages are resolved by thread itself. The workers are ordered in a queue and they are called according to the order in which they were registered. Each worker may perform anything it wants and its running time is not limited. A worker returns a Boolean value after it is done and this value determines if the thread calls another worker or if it terminates current iteration of the main loop. This way a stream of workers can be assigned to a thread and whenever a worker can not finish its task then the workers, which depend on it, will not be called in this iteration. The thread contains many simple methods and some that need a more detailed description:

- `Stop`, `RemoteStop` – both methods are used to stop the threads main function and then it is terminated. The difference is that the remote version stops the thread after all the queued messages are delivered and this is the same for the other remote methods.
- `Sleep`, `RemoteSleep` – the thread is paused for the defined time.
- `Pause`, `Resume` – after the Pause method, the thread is paused until the Resume method is called.
- `WaitForThread` – the current thread is paused until the defined thread(s) is terminated.
- `SendMsg` – all the versions of this method are used to send a message to a thread. The target thread can be defined by its identifier or as a reference to the object that represents it. The message is put in the queue of the selected thread and it is delivered during the next iteration of threads main function.
- `(Un)RegisterWorker`, `(Un)RegisterReceiver` – all these methods are designed as a simplified version of the `SendMsg` method that is used for manipulation with workers and receivers. The method itself constructs the message and sends it to the thread.

- `DispatchMessages` – this method is handles the message delivery. It locks the message queue and delivers every message to its target.
- `ResolveMessage` – the message are delivered to a selected receiver or to the thread itself and this method is responsible for their decoding.
- `ThreadProc` – this is the main function that is used to create each thread. It contains the main loop and during each iteration it calls the `DispatchMessages` method and then it calls all the registered workers.

3.2.6.2. Receiver interface

The Receiver abstract class is an interface that must be implemented by every class used as a receiver. The receivers define the interface of a thread, because they are the only objects that can be used as a target for messages, and they are responsible for message processing. The interface defines only one method – Receive. The derived classes must implement this method and it is called every time a message is delivered to the receiver. The method is invoked by the thread and it is responsible for its complete processing.

3.2.6.3. Worker interface

The Worker interface is used to define the main functionality of workers used in the threads and it is similar to the Receiver interface. The interface requires a single method – Update. The method is called by the thread where the object is registered and it is called once in every iteration of the main loop. The method returns Boolean value and when it returns false, than no other workers will be called and the current iteration will be immediately terminated.

3.2.6.4. Message structure

The Message is a structure that packs all the required message information to a single object. It contains an address of the target thread than there is an identifier of a registered receiver, that will process the message, and than the message contains data used by the receiver to handle the message. The thread reference and the receiver identifier have the same type for every message, but the message data may be different and they may be even empty. The data is derived from a structure MessageData and it is stored in the message when it is created and it may contain anything that the receiver will need to process the message, like a matrix used for a transformation. The data is useful only to the receiver and it is never used during the message delivery.

3.2.7. Windows module

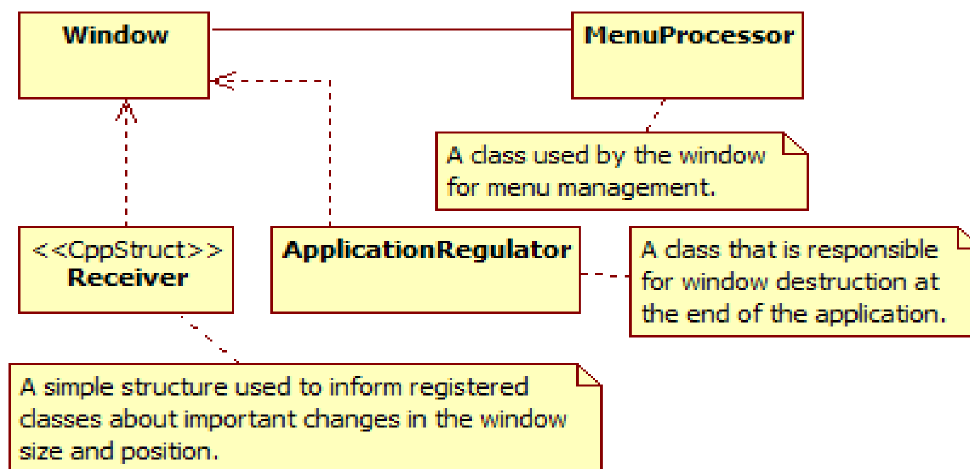

Figure 11: Dependencies between classes in the Windows module.

The Windows module contains classes responsible for windows management and it offers simple interface for window creation and destruction. The Window is a main class of the module and it uses the WindowReceiver for communication with the other objects.

3.2.7.1. Window class

The `Window` class that manages the windows, their creation and destruction, and it is designed as a wrapper for the windows API functions. Its main functionality is concentrated in the constructor and the destructor and the rest is managed by the window procedure assigned to the window. The window procedure is used by the Windows OS to resolve messages sent to the window and it is responsible for the windows behavior. The class contains a default definition of the procedure but it can be provided by a different one when the window is created. Them the class contains a list of objects registered as receivers and if the window is re sized by the user then it informs the receivers that is has been re sized and sends them a message with its new size. It is a very simple class and it defines only a few methods:

- `Window` – the constructor prepares a window class and registers it in the OS and then it creates a new window using the arguments specified by the caller.
- `~Window` – the destructor destroys the window and then it releases its window class.
- `Show` – this method is used change the windows position on the screen, it can show, hide, minimize or maximize the window.
- `Resize` – the method changes the windows size and its status – if it is full-screen or not.
- `(Un)RegisterReceiver` – add or remove an object from the receiver register.
- `GetArea`, `GetClientArea`, `GetBorders` – methods that offer an access to the windows properties - the area and in its different components.

3.2.7.2. WindowReceiver structure

The `WindowReceiver` is a structure that can be registered as a receiver in a window ans it is used by other objects that need to be informed when the window is re sized. It contains the contact information of the objects – the thread that owns them and the identifier of a receiver that will process the message. Then there is defined a specific message data that can store all the required information about the windows new size and position. When an object want to be informed about the changes in the windows size, then it creates an instance of this structure filled with the required information and it registers the instance in the window.

3.2.8. DirectX module

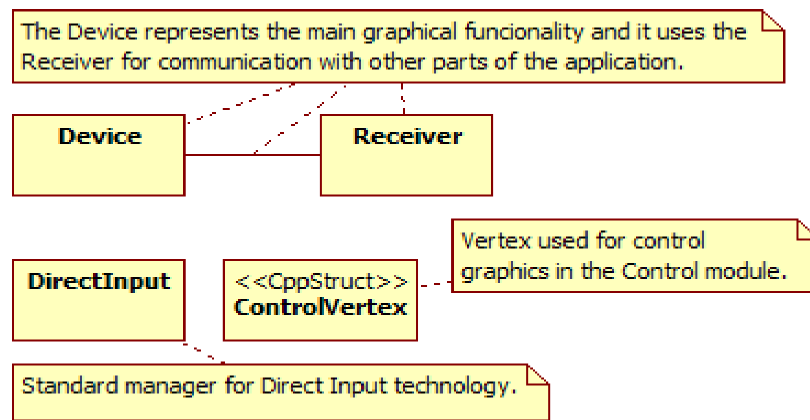

Figure 12: Dependencies between classes in the DirectX module.

The `DirectX` module is constructed using the functionality from the Direct3D library and the Direct Input module and its classes are designed as wrappers that simplify the use of DirectX functions and its classes. The module contains two main parts: 1) classes and functions that create and manage the graphical environment, mainly the Direct3D device and 2) a manager of the Direct Input functionality. These two parts are separated and they are used in different areas of the application, but they are in the same module because they use the same MS library.

3.2.8.1. DirectDevice class

The `DirectDevice` is the main class that manages the entire graphical environment and it provides a special functionality required to keep the device operational. The graphical device is created by the basic Direct3D object of type IDirect3D9 which is obtained from the run-time via a function. The devices final behavior is based on many parameters that are used during its creation and the class must be provided with all the required arguments or at least their default values. The parameters are packed into a single structure for simpler manipulation and the structure is passed to the constructor by the `Graphics` object. The entire creation process is concentrated in the constructor and all the necessary objects are created and stored in the class for later use. It stores the basic Direct3D object, the parameters used to create the device and then it creates and stored the actual Direct3D device. The class has two main purposes: 1) it must create the native device and then provide it to other object that need it for their work and 2) it provides a method that is called before every rendering and it must make sure that the device is operational or the rendering must be stoped. To the first, the class could provide all the functionality that is supported by the native device, but it would be an unnecessary waist, because such methods would just take their arguments and pass them to the original functions. Second, the method packs the entire management of the devices state to a single call because it is the same for every application and it can notify all the registered receivers in the process. The receivers are usually objects that manage some graphical resources, like textures, because these must be reloaded when the device state changes. The main methods are:

- `DirectDevice` – the constructor captures the Direct3D run-time and uses it to create the native device, along with the parameters passed as its argument. Before the device is created, the constructor check if all the parameters are valid and if they are compatible with

the used graphical hardware. If the tests fail then the constructor throws an exception and the device is not created because the creation would lead to fatal error of the application.

- `~DirectDevice` - the destructor is responsible for safe destruction of the device and the Direct3D run-time object.
- `CheckDeviceState` – this method controls the device state and it informs other objects about some important changes. First, it tests the cooperative level of the device and it returns true unless it is in some special state. If it reports an internal error then the method throws an exception, because this means that there may be a serious problem with the hardware. If the device is lost then the method sends a message to every registered receiver and it returns false. And if the device can be reset then it calls the `Reset` method and informs the registered objects and it returns true. The return value is used as a result of the `Update` method of the `Graphics` class and it decides if the rendering continues or if it is terminated using the worker behavior in a thread.
- `AddListener`, `RemoveListener` – objects that need to be informed about the device state are registered using these methods. They must implement the `DirectReceiver` interface.
- `GetDevice`, `GetParams`, `RetRuntime` – these methods allow other objects to access parts of graphical environment.

3.2.8.2. DirectReceiver interface

The `DirectReceiver` is an interface that must be implemented by classes which need to be informed about any changes in the device state and it is the only interface used by the device for communication with other objects. It is useful mostly to objects that manage graphical resources, because they must react to the changes in the device state and this interface is designed specially for the `CheckDeviceState`. The required methods are:

`OnLost` – this method is called by the device when its state is checked and it is lost. The object must release all the graphical resources, that are not managed automatically by the Direct3D run-time.

`OnReset` – the device calls this method before it is reset and the object must reload all the resources that were released when the device was lost, if they are still needed.

3.2.8.3. DirectInput class

The Direct Input represents a very effective input method for applications based on Microsoft technologies because it uses only the DirectX functionality, that works directly with the hardware drivers, and it is available only on the MS Windows platform (and compatible). It creates all the main objects used by the Direct Input library and then it can be registered in a thread as a worker for automated input processing. The required objects are: 1) IDirectInput8 – the basic object used to construct the other, 2) a device for the keyboard input, that is created by the environment object and 3) a device for the mouse that is very similar to the keyboard. All the object are created and they are checked if they are operational before they are used. Their creations is simple and no special parameters are required because their default behavior is almost always good enough. The class is derived from the `ThreadingWorker` and its `Update` method is responsible for the input processing, because the Direct Input provides the information on demand, which is completely different from the standard Windows message-based input method. The class is called every iteration of the main loop and it downloads the information from the devices driver to an internal

buffer – the keyboard uses an array of characters and the mouse uses a simple structure. The main methods are:

- `DirectInput` – the constructor prepares the basic Direct Input object and both devices, for the keyboard and for the mouse. The devices are attached to the drivers and they download the input directly from the driver buffers.
- `~DirectInput` – the destructor safely releases the base object and both the devices.
- `Update` – this method is called in the threads main loop and it updates the input based on the information from the drivers.
- `SetCooperativeLevel` – the input devices can operate under a few different restrictions and this method allows the user to specify if the devices are accessible to the other applications or windows. More can be found in MSDN.
- `GetKeystate`, `GetMousestate` – these methods are the main output of the class, they are used to obtain the input downloaded earlier from the drivers by the Update method.

3.2.9. Control module

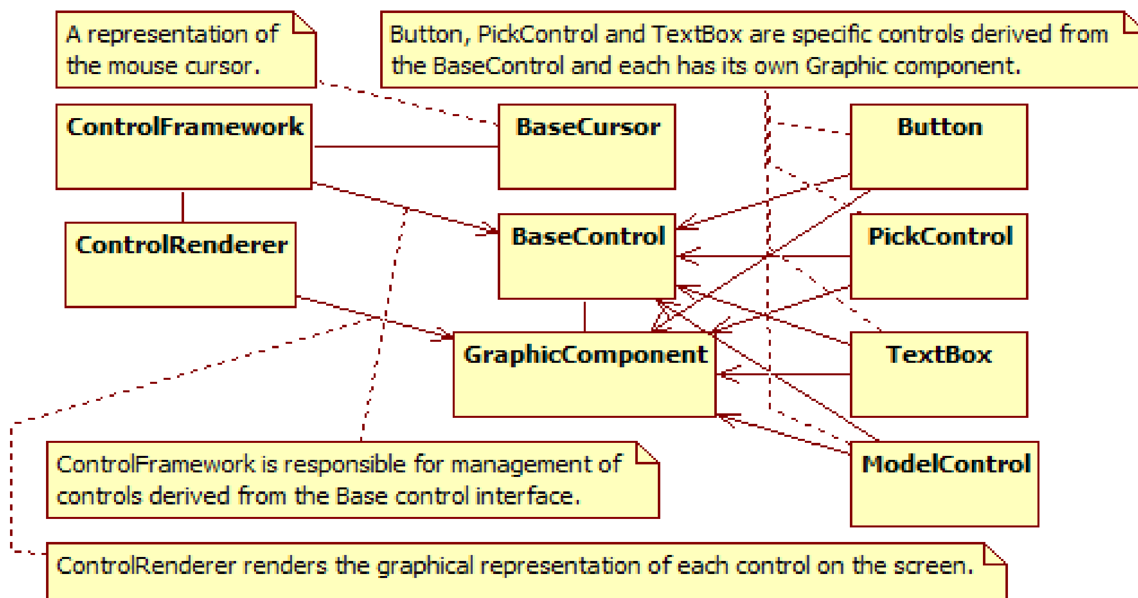

Figure 13: Dependencies between classes in the Control module.

The Control module contains classes that offer complete GUI hierarchy designed for work with the Direct Input technology and its rendering is handled by the functions from the Direct3D library. The module defines interfaces that must be implemented by future controls and these controls are organized in a hierarchy that is part of a framework responsible for input management and processing. The control implement special methods that are called when some input event is encountered and these methods must implement the correct reaction to every input. One control can be selected as a focus control and the keyboard input is delivered to this control instead of any other.

3.2.9.1. ControlFramework class

The main class is the ControlFramework that organizes the controls and processes input obtained from the Direct Input manager. The controls must be registered in a framework before they can be used for any input processing. The registered controls are organized in a tree and their position is determined by their size and position because the invariant is: a control is a parent of all the controls that are placed completely inside it (they may not overlap) and when two controls occupy the same space, then they may be organized in any way. The mouse input is delivered to the control that is under the mouse cursor and it has no children controls that are under the cursor. The keyboard input is delivered to the control that is under the mouse cursor or to the focus control. A control may pass any input it processes to its parent control and this way the input can be delivered to all the controls in the hierarchy that are relevant. The framework is designed as a worker that is managed by a thread and in every iteration it checks the state of the DirectInput object and then it informs appropriate control in the hierarchy. For the rest is responsible the selected control and it may even send the input to any other controls. The framework has six main methods used for manipulation with the controls:

- ControlFramework – the constructor prepares the control hierarchy and it creates a root control, with the same size as the window and with no functionality, that is used as a parent for all the other controls. This way the controls are restricted to the window area.
- ~ControlFramework – the destructor releases the hierarchy data, but the controls are not destroyed. They must be destroyed by their owner.
- RegisterControl, UnregisterControl – standard methods that allow other objects to register their controls to the hierarchy.
- ResolveInput – the method obtains actual input state from the DirectInput object and then it searches the hierarchy tree for the target control. The framework calls a method, of the control, that must process the input..
- Update – this is a standard method used by a threads main loop. It is called every iteration and it simply calls the ResolveInput method.

3.2.9.2. ControlRender class

The ControlRender class is used along the ControlFramework and is is responsible for the presentation of s controls graphics. Every control, that wants to be visible in the scene, must have a special object derived from the GraphicComponent interface and it must register this object in a ControlRender. It is responsible for correct and periodical rendering of the registered objects. The class contains an effect that is used to render the controls and the it is responsible for proper initialization and application of this effect. The class simply allows controls to register and then it periodically renders theirs graphics using the Direct3D device and its own effect.

- ControlRender – the constructor prepares the effect used, later for the rendering, and then it prepares the register for controls.
- ~ControlRender – the destructor releases the effect and the control register, but the controls must be destroyed by their owner (the destructor does not destroy them).
- Receive – a standard method for message processing that is used by the controls to register their graphics.
- Update – this method is called by a thread and it manages the rendering of all the controls.

3.2.9.3. BaseControl interface

The `BaseControl` interface defines the structure of all controls used in the `ControlFramework` because it defines all the methods used to handle events and it provides default body for each method. The default implementation of the methods provides the basic functionality and it defines the standard behavior. The control has some flags that define its behavior when an input event is detected and the methods should process the input according to these flags. The flags define if the control should inform its parent after it has processed an input – the information can be sent down the tree and this is used when a control can not process the input and it lets its parent to handle the event.

- `RegisterControl`, `UnregisterControl` – standard methods that allow other objects to register their controls as children of this control. The registered controls must be placed inside this control.
- `SetFocus`, `NullFocus` – the framework uses this method to inform a control that it was selected as a focus control or that the focus was removed from it.
- `IsPosOn` – the method compares the given point to its size and position and it decides if the point is in its area. This way, the framework checks if control is under the mouse cursor.
- `KeyUp`, `KeyDown` – these methods are used to react on the keyboard input and the derived class should provide its own special implementation.
- `MouseEnter`, `MouseLeave`, `MouseMove` – the framework calls these methods when the user moves the mouse over a control.
- `MouseUp`, `MouseDown` – the framework calls these methods if the user pushes a mouse button when the cursor is in the controls area.

3.2.9.4. GraphicComponent interface

The `GraphicComponent` is an interface designed for work with the `ControlRenderer` class that defines some basic methods necessary for correct rendering. It is derived from the `ThreadingWorker` and the `ThreadingReceiver` interfaces because the render registers the graphical component to a thread and the thread calls the rendering method periodically, every iteration. The controls usually contain an object, derived from this interface, that is registered in the manager and it is responsible only for the presentation of the controls graphics.

- `Present` – this is the main method that performs the actual rendering. Every derived class must place its rendering functionality here.
- `Receive` – method that allows the component to receive messages sent by the control it represents. This is used mainly when the method needs to change its appearance.
- `Update` – this method simply calls the Present method and it is used by a thread where is the object register by the `ControlRenderer`.

3.2.10. Project module

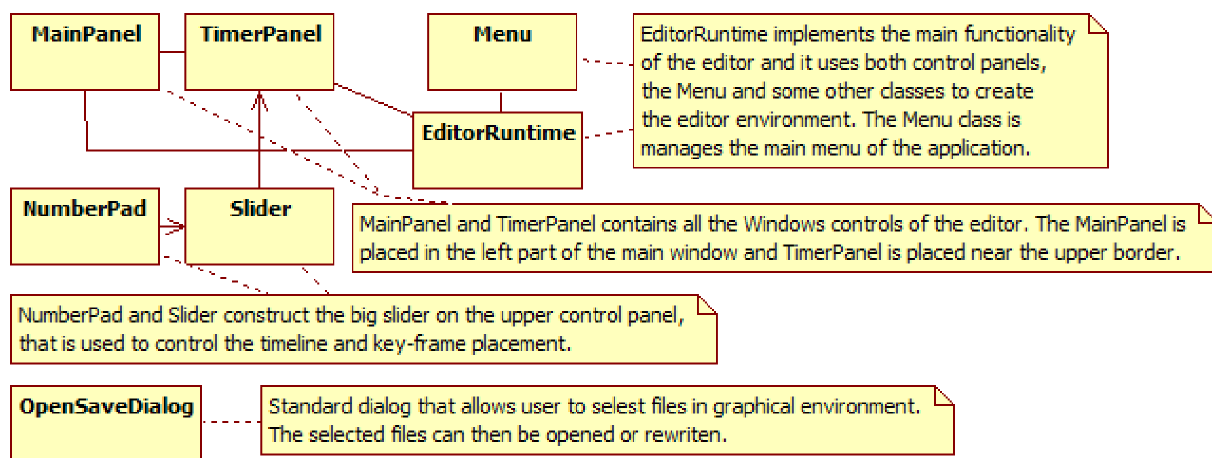

Figure 14: Dependencies between classes in the Project module.

This is the last module and it contains the functionality specific for the animation editor and an implementation of an advanced Windows GUI. The main functionality is provided by the `AnimatedModelAdapter` class and the user interface uses this class to apply the user changes.

3.2.10.1. AnimatedModelAdapter class

The `AnimatedModelAdapter` offers the functionality of the original `AnimatedModel` class but it also provides additional methods that allow other objects to change a models geometry or its animations. The class is designed as an extension of the `AnimatedModel` as it is discussed in the previous section. The adapter uses the internal data of the model to show the changes made by the user, but it hides them from the rest of the application. It contains only private data and the entire functionality is realized by its methods. The adapter contains two copies of the model data to keep track of the user changes and the final state is created from these two copies and stored in the actual model by changing its data provided by the `GetModelData` method. The first copy contains the original data as it was loaded with the model and this copy is never changed. The second copy has the same structure as the original model data, but it contains the user changes and it is changed every time the user does something. This is required, because only this way the editor can provide user with the possibility to make the changes by specifying exact values and it can then show the changes made so far. For example: the user can select a bone and look how much he/she has rotated it compared to the original pose and then he/she can specify an exact angle of the rotation. All the copies must be kept synchronized, mainly when the user adds or removes key-frames of animations. In such case all the copies (even the original data) must be changed in the same way so they can be used without any problems. It is implemented as a part of the `AnimatedModel` namespace in the `Model` module, but it is discussed here because of its specific purpose and functionality. The methods provided by the adapter are:

- `AnimatedModelAdapter` – the constructor creates an instance of the `AnimatedModel` class from a serialized data. Then it creates an effect that is later used for rendering and the constructor obtains handles to its parameters (for easier access to them). This effect implements the actual model animation.

- `~AnimatedModelAdapter` – the destructor releases the animation effect and the internal copies of the model data used for changes.
- `Present` – this method shows a complete application of the skeletal animations. It uses the final bone poses computed by the Animation class and the Present method of the `AnimatedModel` class. It transfers the bone poses to the effect as a parameter and than it calls the Present method of the `AnimatedModel` that renders the models geometry. The effect uses the array of bone poses to animate the rendered vertices, according to the algorithm described in the section 2.4.
- `Pick` – user can modify only the bone that is selected and this method allows him/her to choose a part of the skeleton and modify it later, using the other methods. The bone is selected using a ray intersection with the skeleton and the ray is computed from the mouse position.
- `Rotate`, `Scale`, `Translate` – these methods allow the user to set a new value for any transformation of the selected bone. The changes are stored in a specified key-frame belonging to the bone and the model is later animated using this new transformation.
- `AddKeyframe`, `DeleteKeyframe`, `CreateTake`, `DeleteTake` – these methods create new key-frames and animation takes that are added to the model animation data and the new objects are created as copies of the existing ones. This is the only situation when it is necessary to modify all the copies of the model data.

3.2.10.2. Bones class

The `Bones` class implements the skeleton visual representation. It is used by the model adapter to allow the user to select bones that he/she wants to change. The bones are constructed as a simple 3D shapes, that are rendered using the relative bone poses provided by the `Animation` class. It is described here for the same reason as the `AnimatedModelAdapter` class.

3.2.10.3. Control panels

The `ControlPanels.h(.cpp)` file contains an implementation of functionality required by the Windows GUI that is used in the editor for more complicated operations with the animated model. The controls are organized in two main group and each group is placed on its own panel. Each panel is represented by a class that implements the functionality of all the controls placed on it. The panels are very similar so the implementation is described on the `MainPanel` class.

3.2.10.4. MainPanel class

The `MainPanel` class represents the vertical control panel situate in the left part of the main window (of the editor). The panel is a standard Windows dialog window that must implement its functionality in a window procedure. The class defines a massive static method that manages the input for every control placed on the panel and then it has a few methods used by the window procedure for some advanced operations. The class creates the actual panel as a standard Windows dialog and it sends to the dialog its own address that is used to access its methods in the static procedure.

- `MainPanel` – the constructor creates the native Windows panel dialog and it sends the 'this' pointer to the new dialog.

- `RotateX`, `Scale`, … - these are methods used by the window procedure of the dialog for more advanced operations. The window procedure is very big, because it implements the functionality of every control on the panel, and these methods are used to keep it as small as possible.
- `DlgProc` – the static window procedure used to process the input for all the controls. It contains a big 'switch', that identifies the control targeted by the delivered message and each control has its own part where is the processed the input. If the required operation is simple, then it is implemented directly in the procedure. If the operation is more complex, then a specific method is called using the address specified during the creation of the dialog – the pointer to the `MainPanel` class.

4. Comparison

There are many different implementations of the skeletal animations and this chapter compares the solutions, presented in this project, to some other libraries and books. This chapter also discusses the support offered by the main graphical libraries and it compares the provided support to the implementation presented as the first part of this project – the animation library.

4.1. Main graphical libraries

First, there are many graphical libraries that can be used for rendering and implementation of the skeletal animations and they usually offer some native support. The most significant are technologies implemented for the standard PC platforms and they are divided in two main parts: technologies supported by a single platform and others that can be used on multiple platforms.

The Microsoft DirectX is the main library and graphical platform supported only by the MS Windows OS. It is a very effective and flexible environment that offers many useful functions for graphical applications. This project is built on the standard DirectX library and it can be used as an example of possible implementation (even though it does not use all the classes offered by the DirectX library).

The XNA is a library for the Microsoft .NET platform built on the DirectX that offers some support (described later) for the skeletal animations, but it does not contain an entire implementation. The library contains a definition of data structures that can be used to store an animated model and the content processor is able to load models with skeleton but the actual rendering and animation must be implemented by the actual application. A nice example of functional implementation is described in the book beginning XNA, version 3.0.

The most significant multiple-platform graphical library is OpenGL. This is very old library that is implemented for the MS Winodws and many versions of the Unix OS (Linux, BSD and even the Mac OS X). The OpenGL library offers only minimal support for the skeletal animations and most of the functionality must be provided by the actual application. It supports programmable shaders and basic transformations, but it does not provide any classes that can compute animation transformation and there is nothing that can store the model.

A very specific example is the OpenGE library. It is an open-source graphical library for Sony PSP system and it is very similar to the older versions of OpenGL. The library is pretty simple and it provides only basic functionality like matrix multiplication and rendering using a fixed pipeline. This library does not support any shaders but it offers a very special functionality that can be used to implement a simple skeletal animation. The library allows applications to set up to eight bones and each vertex must contain influence weight (significance) for each bone. This way, a simple model can be easily animated, but it is very difficult to animate more complex models using this method. These special restrictions are present due to the hardware capabilities of the system because it is old (designed around the year 2001) and it is a hand-held console, so it must minimize weight and energy consumption.

4.2. Library support for skeletal animations

Library support is very important for the actual implementation of the skeletal animation and this support is different in every major library. The support is divided in five separate categories for

the purpose of more accurate comparison.

- Shader support – the library must allow programmers to use their own shaders to manage the graphical hardware. This is only possible if the graphical hardware supports the shaders, even though it usually does, there are some exceptions, like the Sony PSP system, where the hardware has only fixed rendering pipeline[25].
- Basic support functionality – the library offers some basic functions that make actual implementation possible. Mainly, the library must allow the users to load the bones to the video memory and use them during the vertex processing, they must be used directly by the hardware. The vertices may be moved in the software, before rendering, but that seriously hurts the performance and such implementation is almost useless.
- Advanced support functionality – the implementation is much more simple when the library offers some utility functions that help with many basic operations, like matrix multiplication. These functions are not designed specially for skeletal animations, but they offer simple solution to many common problems.
- Special support functionality – some libraries offer special functions and data types designed just for skeletal animations and the actual implementation is much easier because many problems are handled by the library functionality.
- Complete implementation – complete implementation includes all the required data structures and functionality, that allows users to simply create an instance of some class, locate the required model and the class renders the model using a selected animation.

The skeletal animation is a complex technique and its implementation must be extensible and it must offer all the necessary functionality. There are some important features that improve the implementation and they make its classes easier to use. The selected features are very important and they are discussed in greater detail in the section 1.1.8.

- Extensible design – many applications need some specific functionality, that is not required for the standard skeletal animations. It is not possible to include everything in the basic implementation, and so it must offer some way to add the functionality.
- Open implementation – open implementation can be important for many users, because it allows them to check how each part works and it may help them to solve many problems.
- Portability – the possibility to use one library on many different platforms is very nice, but it is also very difficult to implement and maintain such library. The implementation presented in this text is based on the DirectX library, but it does not use any advanced functionality and it would be possible to simply modify the implementation so it can be used with the OpenGL library. This modification would include different data types for vectors and matrices and a few simple functions (matrix multiplication and vector transformations).

25 Fixed pipeline does not not allow users to use their own programs (shaders) to process the graphical data, instead it offers simple functions to change the rendering process.

Features	DirectX	XNA	OpenGL	OpenGE	Dragonfly[1]
Shader support	yes	yes	yes	no	yes
Basic support functionality	yes	yes	yes	yes	yes
Advanced support functionality	yes	yes	no	no	yes
Special support functionality	yes	yes	no	no	yes
Complete implementation	no	no	no	no	yes
Extensible design	yes	yes	no	no	yes
Open implementation	no	no	no	no	yes
Portability	no	no	no	no	possible

Table 1: Comparison of the support offered by the main available graphical libraries.

1) – the last column represents the animation library that is part of this project (Dragonfly project). This library offers complete implementation of skeletal animations, but it is not an universal graphical library like the others

4.3. Other open implementations

Books and samples that describe the concepts and implementation of the skeletal animations usually explain most of the facts but they often hide some important parts. This project tries to present a complete implementation of the skeletal animations without any hidden functionality. Perfect example is the book Introduction to 3D Game Programming [Luna06]. It presents very nice and comprehensive implementation of the skeletal animations, but it does not explain how is the animation stored and interpolated (the interpolation is explained, but the code is hidden in some DirectX functions). The examples use functionality from the DirectX library, that is not open-source and users are not able to change the final behavior. This book offers a very good introduction to the programming in the DirectX environment and this project is mostly based on the information from this book.

Then there are open frameworks that offer complete implementation of the skeletal animations, but their solution is either hidden or poorly documented. They usually try to hide as much implementation details as possible to make their use simple. An example can be the Ogre framework that offer complete animated model implementation along with some other more advanced effects and functionality, but the actual implementation is not well documented and the user must study huge number of source files to find how it is done.

5. Conclusion

The project has been successfully implemented and it supports all of the specified functionality. The skeletal animations have been successfully implemented and the implementation have been designed as a separate part of the project and it forms an independent animation library. The animation editor have been completed and it uses the animation library to render animated models. The editor is called Dragonfly (same as the project). It can load and store models in many formats and it offers a wide range of controls for animation editing. Users can create new animations and change their internal structure. Every bone can be positioned as necessary in every part of every animation and this way it allows users to create any desired animation.

5.1. Results

Skeletal animations have been successfully implemented and the result is a complete, stand alone animation library. The presented implementation is comprehensive and completely open and it offers all the required functionality and it can be easily extended. It represents just one possible approach and there can be other, completely different ways how it can be implemented.

The message-based synchronization method turned out to be very complicated because it requires very complicated class design and the entire application must adapt to this design. It is a usable synchronization method, because the application runs smoothly and it is stable, but the difficulties caused by the message-based communication are too great for it to be used in commercial applications. This technique may be effective when the communication between threads is very simple or at least if the messages does not need any response. Otherwise, it is better to use the standard lock-based synchronization and there are already some lock-free algorithms, that can provide a safe access to shared memory.

The Dragonfly editor uses the implementation of skeletal animations presented in this project and it adds required functionality as an extension of the basic classes. The editor presents an actual use of the classes and it shows that they provide sufficient functionality and flexibility. It is implemented as a multithreaded application and the access to the shared memory is managed by message-based synchronization. It runs without any problems and it is fast and responsive, even when the user does complex changes to the model animations.

6. Appendix A

This chapter describes the content of the accompanying DVD. All the projects, that are part of the DVD, were created in MS Visual Studio 2008. All the projects use DirectX SDK (August 2009).

Folder	Content
DirectX SDK	The folder contains DirectX SDK used to build all executable files in the project. The folder contains plugin for Autodest 3DS Max that can export MS X files (native format of DirectX). The last file is a patch for the Visual Studio that makes linker more stable when building big projects.
Documentation	Documentation generated from the source codes. The original HTML documentation can be found in the **html** folder. Then there is a compiled version of the documentation.
Dragonfly Editor	Source files and complete executable files of the Dragonfly animation editor. The complete distribution can be found in the **bin** folder.
Dragonfly Library	Source files of the animation library. The compiled static library can be found in the **bin** folder and it is available in release and debug version.
Import Export Libraries	Source files and compiled libraries used by the Dragonfly editor to load/store animated models. The compiled DLL libraries can be found in the **bin** folder.
Texts	Texts of the project: this text(MainText.pdf), **User Documentation** (UserManual.pdf), Specification (Specification.pdf), Abstract in Czech and English (Abstrakt.pdf and Abstract.pdf)
Tutorials	Source files of all the tutorials. The tutorials are described in the user documentation. Compiled version of the second tutorial can be found in the **bin** folder.

Table 2: Content of the accompanying DVD.

List of Figures

Figure 1: An example of morphing animation. The two extreme expressions are combined to create the intermediate ones. The image is from [Granberg09]......7
Figure 2: The skeleton binding. The image is from [Luna06]......8
Figure 3: Three key-frames of a character and its skeleton. Additional poses are computed from these key-frames based on current time. The image is from [Granberg09]......8
Figure 4: Description of symbols used in the diagrams to represent a relationship between classes. The only exception is the dashed line that is used to connect the notes with their targets......26
Figure 5: Dependencies between classes in the Model module. This module contains all the classes and files necessary to use the skeletal animations as an independent library. The module contains classes that are not part of the animation library, but they are closely related to the library and they work with the skeletal animations too......27
Figure 6: Dependencies between modules in the project......33
Figure 7: Basic relations between modules including the common data exchange and the most important classes......33
Figure 8: Dependencies between classes from the Runtime module and the Project module......35
Figure 9: Dependencies between classes in the Timer module......39
Figure 10: Dependencies between classes in the Threading module......39
Figure 11: Dependencies between classes in the Windows module......41
Figure 12: Dependencies between classes in the DirectX module......43
Figure 13: Dependencies between classes in the Control module......45
Figure 14: Dependencies between classes in the Project module......48

List of Tables

Table 1: Comparison of the support offered by the main available graphical libraries......53
Table 2: Content of the accompanying DVD......55

Bibliography

DeLoura00: Mark DeLoura, Game Programming Games, Vol. 1, 2000

Granberg09: Carl Granberg, Character Animation with Direct3D,Course Technology, 2009

Luna06: Frank D. Luna, Introduction to 3D Game Programming with DirectX 9.0c—A Shader Approach, 2006

Möller02: Möller, Tomas and Eric Haines, Real-Time Rendering. 2nd ed. , 2002

MSDN1: Direct3D mesh interface description, http://msdn.microsoft.com/en-us/library/bb174069(v=vs.85).aspx

MSDN2: Direct3D vertex buffer description, http://msdn.microsoft.com/en-us/library/bb205915(v=vs.85).aspx

MSDN3: Direct3D index buffer description, http://msdn.microsoft.com/en-us/library/bb205865(v=vs.85).aspx

MSDN4: Description of states of the Direct3D graphical device, http://msdn.microsoft.com/en-us/library/bb324479(v=vs.85).aspx