VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

GRAFICKÉ INTRO DO 64KB S POUŽITÍM OPENGL
GRAPHIC INTRO 64KB USING OPENGL

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE AUTHOR	PAVEL GUNIA
VEDOUCÍ PRÁCE SUPERVISOR	Ing. ADAM HEROUT, Ph.D.

BRNO 2007

Abstrakt

Tato práce pojednává o fenoménu grafického intra, často označovaném jako digitální graffiti. Detailně se rozebere téma grafického intra s omezenou velikostí a popíší se techniky vhodné k jeho realizaci. Na konci práce jsou diskutovány postřehy a zkušenosti získané během tvorby, stejně tak i celkové shrutí a pohled do budoucna.

Klíčová slova

počítačová grafika, digitální umění, demo, intro, OpenGL, procedurální generování, komprese, parametrické křivky, parametrické plochy, syntetizovaná hudba

Abstract

This work deals with the phenomenon of graphic intros, a digital graffiti of the modern age. The focus is put on size restricted animation of size of the executable file lower than 64 kilobytes. It reveals the main techniques used. Finally, interesting aspects and experiences that came up are discussed, as well as the conclusion and future work proposal.

Keywords

computer graphics, digital art, demo, intro, OpenGL, procedural generation, compression, parametric curve, parametric surface, synthetized music

Citace

Pavel Gunia: Graphic intro 64kB using OpenGL, diplomová práce, Brno, FIT VUT v Brně, 2007

Graphic intro 64kB using OpenGL

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Ing. Adama Herouta PhD.

....
Pavel Gunia
May 22, 2007

Contents

Chapter 1

Introduction

The purpose of this thesis is to present a subculture of computer art, dating from the late 1970's but still rather not well known to the general public. This work, in particular, speaks about short animations, furthermore about size-restricted to 64KB. A part of the project was producing a final piece of computer art, a short animation. However, the focus is put on describing techniques chosen or developed. The primary intention was to create a library and a set of tools for possible future use or as a inspiration for others.

1.1 Graphic demo and intro

A *graphic demo* is a noninteractive multimedia presentation or animation. In contrast to the common video, it is produced as an executable program, which renders the animation in real time, making the computational power of various computing devices a considerable challenge. There are several categories demos are informally classified into, based on the output platform, size restriction or contents of the demo. Nowadays, it is the most common to see demos running on Microsoft Windows, as a stable and uniform platform providing good support for hardware accelerated rendering. However, there is a wide range of other popular platforms, such as Linux or BeOS or text mode animation. The challenge of real time animations eventually spread on some curious platforms as PDAs, mobile phones, iPod, Nintendo or calculators, extending the horizon of possible applications of those devices.

Another challenge is considered the size restriction of an executable code of a demo. There are several categories, usually based on technological capacities of older computing devices. These limitations are in most cases obsolete with modern computers, but they provide a competition area for coders. A size limited demo is often referred to as an *intro*. The most important is the division between the "full-size" demos and the size-restricted intros, a difference visible in the competitions of nearly any demo party. Because of the strict size limits, intros show off the programmer's ability to squeeze much into little space, often by generating graphic and sound data rather than just reading it from a datafile. Because of the extremely low size limit, 4K intros used to lack sound, or had extremely low quality music. Some demoparties organize 1K, 256 byte or even 64 byte intro competitions. While creating a 4K might not require low-level programming knowledge anymore, less-than-1K competitions require the demo coder to be skilled in both assembly programming and algorithmic optimization. The most typical competition categories for intros are the 64K intro and the 4K intro, where the size of the executable file is restricted to 65536 and 4096 bytes, respectively. It is also quite common to classify demos by style and content rather than technology. Storydemos, for example, are based on a story line, while ravedemos share the musical and visual aesthetics of rave parties. The most experimental, unusual and controversial demos are often referred to as art demos or abstract demos. Many groups have a distinctive style of their own, and sometimes a demo can be described by referring to a well-known group cultivating a similar style.

Figure 1.1: A 64K demo *Chaos Theory*(2006) by *Conspiration* ranked 2nd at *Assembly 2006* party.

Demo making is considered a subculture of computer art, referred to as a *demoscene*. The animation itself is rarely made by a single person, instead a *demogroup* is formed, involving programmers, musicians and graphic artists. Members of a demogroup, *demosceners*, usually share similar ideas and the group keeps their characteristic style of animation. A demoscener is typically specialized in a certain area of creativity. The traditional division is in coders, graphic artists and musicians, who are specialized in programming (often including overall design), still graphics (including 2D art and 3D modelling) and music, respectively. Eventually, groups compete with each other in technical and artistic excellence. That takes place either on the internet or, more likely, on a *demo party*. A party based competition, so called *compo*, is basically a huge LAN party held for several days. Demogroups enrolled in the competition usually present their work since the last demo party. However, depending on the rules of the competition, it is not rare to compete in making demos straight on the party. There are usually no restrictions on the contents of competing demos. The vote is public and generally the overall impression decides.

1.2 Development

The demoscene dates form the 1980's, in the era of 8-bit computers such as Commodore 64 or ZX Spectrum and becomes widely popular with the rise of 16/32 bit computers, as Atari or Amiga. In the early years, demos were strongly connected with software hacking.

The cracker or the team commonly inserted short graphic introductions in order to take credit for their effort. Later, the scene evolved into a standalone culture independent from software piracy. These demos usually involved scrolling text or bitmaps, simple vector graphics and synthetised music. Competitions and parties were held at own houses or high school gyms, involving only few people. Later, in the nineties, more complex vector routines comes, bringing the importance of three dimensional approach. Larger parties are organized in town arenas of major cities. The advancement of hardware accelerated graphics brings PC to the front of interest.

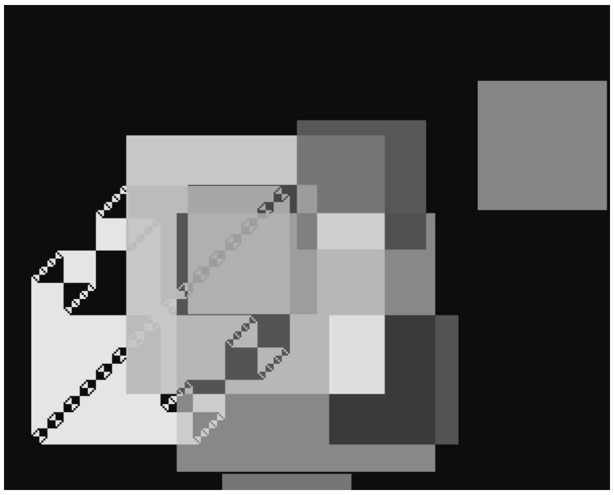


Figure 1.2: A still image from *munching squares* display hack. Image taken from [4].

1.3 Czech and Slovak demoscene

The czechoslovak demoscene dates from early nineties with *GIH DemoBit demo-party (1993)* in Bratislava, held as a competition of high school projects. Lack of serious demoscene in the Czech Republic brought czech demosceners to the idea of organizing *Fiasko demo-party (1998)* in cooperation with the Charles University in Prague, Faculty of Mathematics and Physics. Raise of the czech demoscene is clearly seen in those years. Demos and intros submitted to international competitions brought fame to czech programmers and computer artists, especially in the category of extreme size restriction such as 256 byte or 4KB.

1.4 Motivation

Even through the extensive technological progress in computer graphics, today's demoscene is, generally, divided in two contrary parts. One is a bit more than a mere audio-digital projection, while the other is rather blindly following the eye-taking technological modern trend. We can see creations whose high artistic spirit will outlive all the new technological excesses of modern time. The other, however, often creates an imaginary sieve where many brilliant pieces might finally end unnoticed. That is rather believed proposal of the mainstream demoscene further development.

This project relations to the Year Project in some aspects. It employs knowledge of computer graphic techniques and approaches, specifically related to OpenGL standard. Skills obtained through the last year work on the Year Project were used to tailor selected techniques to produce a minimalist animating solution. We try to create a functional set of tools for further animation development. The primary focus is not on technically advanced rendering, however, several recent techniques and optimizations are considered to be implemented, to provide additional space for someone's creativity.

Chapter 2

Concept

The primary intention is to develop a solid system for creating short, size-restricted intros. Basically, it is two major units that together make an intro. It is a rendering system and data of the scene. Each demogroup has usually its own *demotool* tailored to the approach the group takes towards the process of animation-making. Intros are executable programs, and the program code created by the coder is still considered a very important element of an intro. Although there are programs known as demomakers or demotools that allow the creation of technically decent demos without coder involvement, demo groups not using any code of their own are often considered as "cheating" the demo making. Furthermore, in order to get an in-depth understanding of what is behind making an intro, no third-party components will be used.

The work will be divided into 3 major units. Firstly, it is *DemoBasic*, a library with a minimalist rendering system and structures to describe and animate the rendered scene. Secondly, it is a demotool application. Finally, an animation is created using the components mentioned above. See fig. 2.1.

2.1 OpenGL is a good choice

The rendering system will be built on OpenGL graphic library, the industry's widely used and supported 2D and 3D graphics application programming interface(API). It provides a basic API to hardware accelerated graphic routines, is well arranged and easy to use as a graphic foundation for higher-level APIs. Furthermore, OpenGL is supported on all UNIX® workstations and shipped standard with Windows and MacOS personal computers. It runs on all major operating systems including Windows 95/98/2000/NT/XP/Vista, MacOS, OS/2, UNIX, LINUX, OPENStep and BeOS [8][9]. At present, the rendering system and the animation itself is being developed for PC running Microsoft Windows. However, portability to other platforms has been kept in mind as a possible future development of the project.

The rendering system, which forms the DemoBasic library, will be composed as a set of data structures describing entities of the animation and functions operating with them. Considering the size of the output code, it is primarily designed as an automation of OpenGL commands, data structures and general techniques. On top of that are methods providing

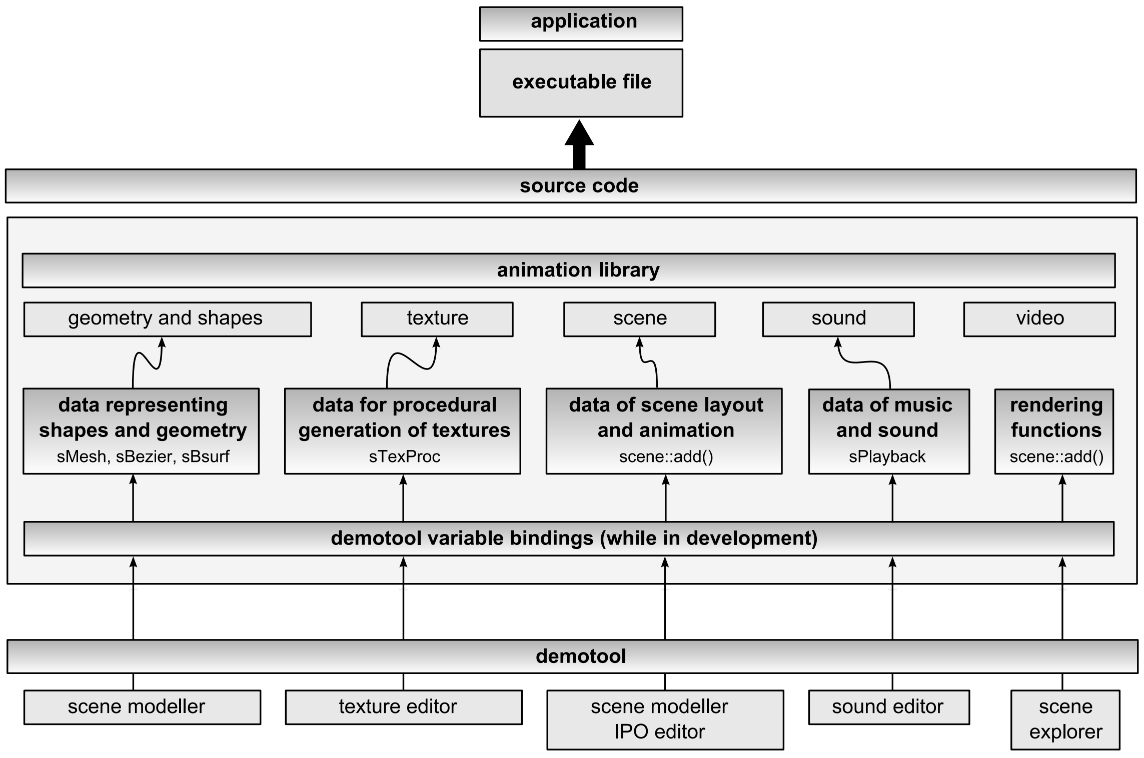


Figure 2.1: A model of an approach.

secondary functionality including decompression of data, parametric definition of data and scripting.

Let's begin with the basic functionality.

- defining and showing geometry
- lighting of the scene
- texture states and materials
- environmental attributes
- handling OpenGL extensions

2.2 Basic functionality

This section describes the basic entities and methods designed for DemoBasic library. They are intended to provide an easy-to-use approach to general 2D/3D rendering techniques and OpenGL commands [14]. Furthermore, they prepare solid ground for further approaches more relevant to intro making and to the size restriction.

2.2.1 Geometry

Geometry data will be held in indexed data arrays. That is, data such as vertices or normals are held in separate sequential buffers, index buffer provides references to enabled buffers when constructing geometry primitives. This is an approach identical to OpenGL routine *DrawElements*[13]. This approach, in contrast to direct construction from data arrays, provides a compact way of storing data.

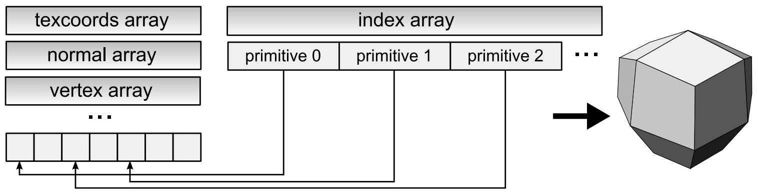


Figure 2.2: Constructing geometry via indexed arrays.

The focus is primarily on the simplicity of code and compatibility with different graphic devices. Vertex arrays, included in OpenGL 1.1 Standard, provide a compromise between compatibility and rendering performance. A favourable alternative to vertex arrays is *compiled vertex arrays*, defined through *EXT_compiled_vertex_array* extension. It allows the vertex buffers to be locked, giving the driver more optimization opportunities. Another option is to use *Vertex Buffer Objects*, defined through *ARB_vertex_buffer_object* extension. This mechanism allows various vertex data to be cached in high-performance graphic memory on the graphic device, therefore significantly increasing the data transfer rate. However, older implementations of OpenGL do not support vertex buffer objects. See section 2.2.3 about OpenGL extensions.

OpenGL evaluators

It is intended to widely employ parametric surfaces, particularly Bézier surfaces. OpenGL natively supports rendering of polynomial surfaces with Bézier basis through a mechanism called *evaluators*. For various reasons, evaluators have not been particularly popular as an interface for drawing curves and surfaces. Generally speaking, OpenGL evaluators provide only basic functionality for rendering of surfaces and there are several major limitations. Firstly, due to the nature of OpenGL, there is no straightforward mechanism to retrieve the geometry values produced by evaluators. If it is the case the geometry data is needed for further computation (case of collision detection of objects for example), OpenGL evaluators can not be used.

Secondly, the basic functionality of evaluators does not cover the needs of today's modern approaches and techniques. OpenGL evaluators provide a way to use polynomial or rational polynomial mapping to produce vertices, normals, texture coordinates and colors. It is, however, desirable to use other vertex attributes in nowaday's techniques including multiple texture coordinates sets for multi-texturing, vertex weights for skeletal animation, tangents and binormals for custom per-pixel lighting and others. See section 2.2.3. This obstacle would be partially solved by a specific OpenGL extension, particularly *GL_NV_evaluators* by NVIDIA Corporation, proposing a new interface for surfaces that provides a number of significant enhancements to the functionality provided by the original

OpenGL evaluators [10]. This extension is, however, not longer supported in driver updates after November 2002 due to low popularity of evaluators.

Furthermore, only the number of rows and columns can be specified for the tesselation process [13]. This does not allow optimizing the tesselation for particular purposes, such as progressive resizing of the tesselation grid based on the distance from viewer. Also, difficult problems can arise when multiple adjacent patches are drawn. Numerical accuracy problems can cause cracks to appear between patches with the same boundary control points.

At last, evaluators involve a lot of math. Many implementations do not optimize evaluators and the performance becomes unacceptable in immediate mode. Furthermore, inserting evaluators in a display list does not cause any significant performance increase, that is because evaluators themselves being compiled into the display list are not pre-calculated to produce the geometry data. See fig. 2.3.

It is a commonly proposed approach to implement own tesselation routines. See section 2.3.

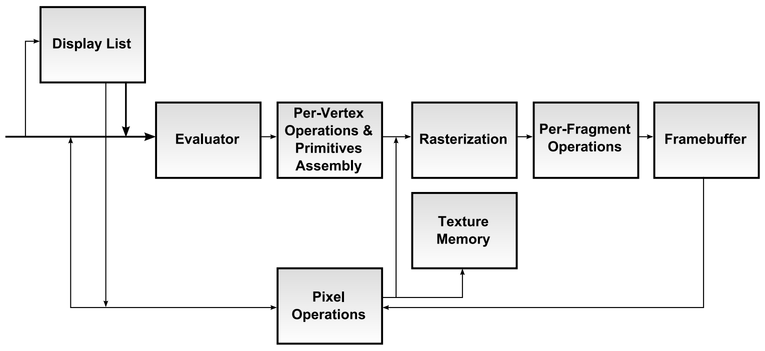


Figure 2.3: A simplified diagram of OpenGL rendering pipeline.

Structure

The structure itself is intended as a foundation for procedural-generated geometry or parametric surfaces. It proved to be useful to contain binormals and tangents in the structure. A rather high amount of custom per-pixel lighting is expected in an intro. (see section 2.2.3.)

Structure for defining geometry will allow following data.

- vertices – essential for geometry construction
- normals
- binormals
- tangents
- multiple texture coordinates – a set of texture coordinates for each texture layer

- vertex colors
- vertex groups and bones

2.2.2 OpenGL state - lighting, materials and textures

OpenGL, internally, acts as a state machine. That is a collection of states that holds the information about current lighting parameters, material definitions and many others. Then, the rendering of each primitive drawn is affected by the current state. OpenGL, naturally, provides mechanisms to query and control the current state. The goal is to automate state changes of logically separate units including lighting, texture state definition and material properties. To do that, we will hold the data relevant to each unit in a structure, allowing to change the state to reflect the given data in one function call.

Light

The structure will hold data relevant to defining a light source as specified by OpenGL function *glLight*[13].

Material

The structure will hold data relevant to defining a material properties as specified by OpenGL function *glMaterial*[13].

Fog

The structure will hold data relevant to defining a fog properties as specified by OpenGL function *glFog*[13].

Texture state

This is a slightly different case. The structure will hold data relevant to texture binding and texture combiners settings for one or more texture units (see section 2.2.3). Furthermore, the routine responsible for applying the changes will guarantee the consistency of the OpenGL state. That is, primarily, checking the limits of current hardware, enabling all used texture units and disabling the unused ones. Furthermore, the implementation will optimize OpenGL state changes.

2.2.3 OpenGL extensions

OpenGL is architected for flexibility and differentiation. The processing pipeline is defined by the OpenGL specification, however, platform vendors have the freedom to tailor the OpenGL implementation to meet desired requirements. That is achieved by extending the OpenGL specification. Generally speaking, an OpenGL extension is a functionality that is not (or is different) in the original specification. Each extension is uniquely identified by its name and defined by its specification [5]. Basically, an OpenGL extension brings either a whole new function or an alternative to behaviour of an existing function.

OpenGL Architecture Revision Board *(ARB)* is an independent consortium governing the future of OpenGL. Its goal is to approve OpenGL specifications and advance the standard. In July, 2006, it has joined with the Khronos Group, an industry consortium focused on creation of open standard APIs [11].

The fundamental OpenGL specification (version 1.0) is guaranteed to be supported by every OpenGL-compliant graphic device. In today's rapid development of computer graphics industry, graphic hardware vendors encapsulate their developed techniques and improvements in OpenGL extensions providing access to cutting edge rendering functionality so the application can achieve higher performance and rendering quality. In addition to opening the door to the latest features of the hottest new graphic hardware, it provides backward compatibility strategy with older OpenGL implementations which does not support the desired extensions. OpenGL employs a mechanism to determine at run-time whether the OpenGL implementation supports the particular extension or not. Furthermore, each time the standard advances, approved extensions are included and the compliance of a graphic device with an OpenGL standard (eg. 1.1, 1.2, 1.2.1, 1.3, 1.4, 1.5, 2.0, 2.1) guarantees the presence of support of all extensions included in the standard.

Due to the demanded size limitation of the rendering system, ARB approved extensions are recommended. They provide functionality respected by majority of hardware vendors and graphic devices, hence it is not necessary to develop individual routines specific to the design of current hardware. An important issue is to provide functionality with graphic hardware that does not support the specified extension.

Structure

The library will contain a set of functions to work with OpenGL extensions. Firstly, it is a tool to query the support of the extension on current hardware. Secondly, it is a set of routines which provide access to the extension's routines. The library will contain mechanisms to:

- query the presence of given extension on current hardware at run-time
- provide entry points for routines implemented by given extension

Furthermore, we will provide automation of routines specific to frequently used extensions. The attention was put on simplifying the use of techniques relevant to modern approaches. Namely, the decision is:

- to optimize performance of rendering of geometry using *EXT_compiled_vertex_array* and *ARB_vertex_buffer_object* extension
- easy use of multiple texture units for multi-texturing via *ARB_multitexture* extension (included in the OpenGL 1.2.1 Specification)
- accessing programmable texture combiners through *ARB_texture_env_combine* and *ARB_texture_env_dot3* extensions (included in the OpenGL 1.3 Specification)
- to allow low-level custom vertex programming with *ARB_vertex_program* extension (OpenGL 1.3 Standard is required)
- to allow low-level custom fragment programming with *ARB_fragment_program* extension (OpenGL 1.4 Standard is required)
- to access advanced image processing capacities of the graphic device via *ARB_imaging* extension (included in the OpenGL 1.2.1 Specification)
- *ARB_point_sprite*

Optimizing rendering of geometry

Compiled vertex arrays is a simple technique allowing the user to lock portions of the vertex data. At that step, driver can perform several optimizations on them including removing duplicated vertices and copying the data to high-performance memory. Going further, the need of copying the data can be removed using vertex buffer objects.

Multi-texture and texture combiners

Multi-texturing is a frequently used technique. It allows to process and apply multiple textures to rendered geometry in a single pass. The ARB_multitexture extension implements several functions. In order to keep the size of code as small as possible, we will use the necessary minimum:

- *MultiTexCoord2f* and *MultiTexCoord3f* allow to define two and three-dimensional texture coordinates for given texture unit
- *ActiveTextureARB* and *ClientActiveTextureARB* allow to select the active texture unit for server and client side respectively

Programmable texture combiners is a mechanism allowing further control over blending operations when multi-texturing is employed. It provides limited facility to per-pixel operations in contrast to fully programmable fragment programs, however, most implementation optimize the performance of texture combiners making them noticeably faster. Neither ARB_texture_env_combine nor ARB_texture_env_dot3 introduce new functions, however, behaviour of OpenGL standard *TexEnv* function is expanded to reflect the new functionality.

Vertex and fragment program

Vertex and fragment programs expose a significant degree of per-vertex and per-fragment programmability, making the vertex and fragment operations highly customizable. For the sake of simplicity and size of the code, *ARB_vertex_program* and *ARB_fragmen_program* are used for GPU programming. Basically, vertex and fragment program is a sequence of floating-point 4-components vector operations written in assembler-like programming language, compiled at run-time and loaded into the graphic device.

Imaging subset

The new features of the imaging subset are primarily intended for advanced image processing applications. It is, however, possible to employ this functionality to perform post-processing filtering in order to enhance the quality of real-time rendered scene, hence, to provide impressive effects.

The imaging subset brings following additional functionality to standard pixel operations.

- **color tables**
- **color matrices**
- **convolution** - 1D, 2D and separable

- **pixel statistics**
- **new blending equations**

Point sprites

Specific tasks, such as rendering of particle systems, need the geometry to face the viewer at all times. The conventional approach required an additional state change to remove the rotation of the viewer from transformation matrix before each primitive is drawn. Furthermore, particle systems have tended to use a four-sided rectangular polygons to render their geometry. That can be expensive on vertex-processing operations for systems with a high number of particles as this approach quadruples the amount of geometry needed to be processed. The purpose of *ARB_point_sprite* extension is to allow to render particle systems using standard OpenGL points rather that quads. The new functionality provided includes texture-mapped rectangular points and automatic generation of texture coordinates.

2.3 Parametric curves and surfaces

When Archimedes (287 BC – 212 BC), a Hellenistic mathematician, physicist, engineer, astronomer, and philosopher, introduced his inventions for irrigating high-pitched areas, he defined what is believed to be one of the first mathematical curves, Archimedes' spiral. The spiral is a set of points corresponding to the locations over time of a point moving away from a fixed point with a constant speed along a line which rotates with constant angular velocity. Nowadays, scroll pumps and scroll vacuum pumps as well as some DLP television sets still use technology based on Archimedes spiral. Johannes Kepler applied various curves in astronomy, clearly in his study of Mars' orbit relative to the Earth.

With the industrial development in the nineteenth century came the need for more complex analysis and stable solutions. New production technologies demanded a method to describe surfaces and shapes, accurate and still suitable for machine processing. Pierre Étienne Bézier (1910 – 1999), a French engineer, developed a mathematical solution of the *Bézier curves* and *Bézier surfaces* for Renault, where he was working on UNISURF CAD CAM system. In 1959, Paul de Casteljau, a physicist and mathematician at Citroën, developed an algorithm for computation of a Bézier curve. It has been widely used, although the efficiency has been questioned compared to today's algorithms.

There are two major ways of seeing curves and surfaces, based on the way they are defined and their purpose.

- An *interpolation curve* and an *interpolation surface* is a tool for modeling shapes and objects of the real world. A Bézier curve and Bézier surface is an example of employing curves to design shapes of objects in the modern industry. An interpolation curve and surface is defined by the set of points in space which it goes through. Saying in simple words, they are defined by their appearance.
- A *mathematical curve* and *mathematical surface*, on the other hand, represents the result of a mathematical formula. Therefore, it is used as a visualization of the result of a computation.

Interpolation curves and surfaces offer convenient way for modeling shapes. Basically, a provided set of points defines the shape. A continuous and smooth curve or surface is put through these points. It has the capacity to interpret basically any shape, and yet provides an intuitive approach to the modeling.

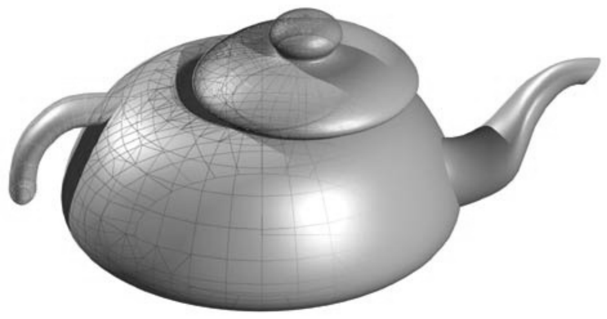


Figure 2.4: A teapot constructed using parametric interpolated surfaces.

2.3.1 Bézier curve

Bézier curves and surfaces are now used in most computer-aided design and computer graphics systems. There is robust and stable mathematical base and efficient algorithms. They represent smooth shapes, which are easy to manipulate, have good continuity properties and require relatively small amount of input data. In addition, common parametric shapes such as spheres, cylinders or cones can be well approximated by a small number of Bézier surfaces. Rational Bézier curve adds adjustable weights for each control point to provide closer approximations to arbitrary shapes.

Application

In real situations, Bézier curve is considered in one, two or three-dimensional Euclidean space. However, there is no limit on the dimension in which the curve is evaluated. Computing is componentwise, therefore the calculation is separable for each component of the vector. For the reason of effectivity and computation speed, at most cubic Bézier curves, of degree 3, are used. More complex shapes are constructed with several curves joined together. This method is known as a *spline*, in this case a Bézier spline.

There is a wide variety of application for Bézier curves. Originally, two and three dimensional curves were designed to represent shapes. Furthermore, they are well suited for defining paths in space, which are traced by objects. The movement is smooth and continuous, naturally, Bézier splines allows for sharp corners at points where two curves meet. One-dimensional Bézier curve is also interesting. It can be used as an interpolation function for a variable changing it's value over time. This can be beneficial when defining an animation of an object.

2.3.2 Bézier surface

Bézier surface can be viewed as a generalization of Bézier curves. It can be of any degree, but cubic Bézier surfaces generally provide enough degrees of freedom for most applications.

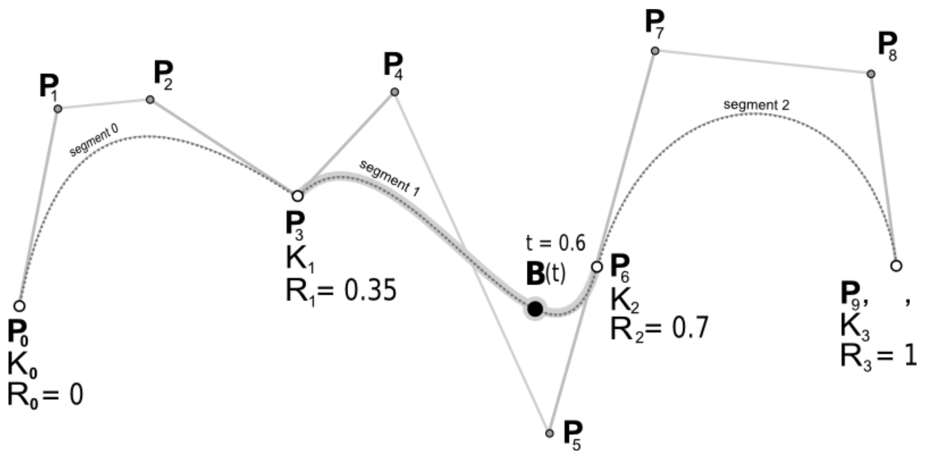


Figure 2.5: An example of a Bézier spline constructed from three Bézier curves.

A Bézier surface of degree (n, m) is defined by $(n+1)(m+1)$ control points $\mathbf{P}_{i,j}$ for integer indices $i = 0$ to $n, j = 0$ to m.

2.3.3 Implementation

Due to their complexity, Bézier curves and surfaces can not be rendered directly. Instead, they are transformed into a set of primitives understandable for graphic hardware. The transformation is included as core function in OpenGL standard through one and two dimensional *evaluators*. Given a set of control points, degree of the polynomial function and interval on which to evaluate, the curve or surface is rendered using OpenGL primitives. This approach clearly benefits from the computational power of the graphic device. On the other hand, due to the nature of OpenGL, it is not possible to store the calculated data into any kind of buffer for further reading. This is a considerable disadvantage, as the curve or surface must be evaluated every time it is rendered. It is also desirable to keep the evaluated coordinates for further processing or calculations, for example animating the structure of an object or creating interactions and employing physics. Obvious case would be a Bézier curve, which is rarely rendered, instead it is mostly used as a path for animating movement of objects.

2.4 Noise and procedural textures

2.4.1 Perlin noise

Perlin noise was developed in 1981 – 1983 by *Ken Perlin* as a part of his study of procedural textures for *Mathematics Application Group, Inc. (MAGI)*. At that time Ken Perlin worked on special computer imagery for *TRON* (1982), the first movie with a large amount of solid shaded computer graphics. In 1985 the noise was presented as a paper at *SIGGRAPH 85 Annual Conference*[2]. He revised and improved the algorithm in 2002, which was described at SIGGRAPH 2002 paper [3].

The main aim of Perlin noise is to produce naturally-looking distorted signal. In the nature, many things are in the form of fractal. They have various levels of detail. Perlin noise follows the same idea, it is an interpolation of multiple noises of different frequencies as shown in figure 2.6. Perlin noise is widely used in computer graphics for effects like fire, smoke, and clouds. It is also frequently used to generate textures when memory is extremely limited, which makes it well suited for graphic intros. However, Perlin noise can be applied in basically any case where a smooth natural-looking distortion is desired.

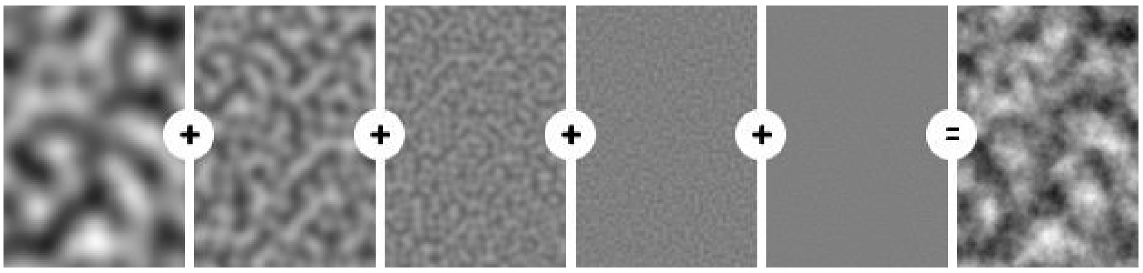


Figure 2.6: A principle of Perlin noise. Five layers of noise of various frequency and amplitude are blended together.

The gist of Perlin noise is a *noise function.* For given frequency, the noise function generates a smooth *noise wave*, which is used to create a layer of Perlin noise. The original approach, as developed by Ken Perlin and described in his paper [2], employs cubic polynomial function generating the wave from given random values. However, several other methods have been described, considering the speed and efficiency of the process.

The noise function

The essential part is a random noise generator. In this case, it is desirable to employ a generator based on a seed, so that it is possible to reproduce the same noise again. The noise function operates on a domain of real numbers and can be of basically any dimension. A random value is generated at each multiple of the given wavelength. To produce smooth noise, the interval between two generated values is interpolated. There are three major interpolating techniques.

Linear interpolation is the simplest technique. The algorithm can be possibly used for real-time computation, the drawback of this method is clearly the poor quality of the result.

Definition 2.1 Given points $\mathbf{A}$, $\mathbf{B}$ and the fraction f of the interval between them, $f \in [0,1]$, linear interpolation $\mathbf{I}_{LIN}$ between $\mathbf{A}$ and $\mathbf{B}$ at position f can be defined as

$$\mathbf{I}_{LIN}(\mathbf{A},\mathbf{B},f) = \mathbf{A}\ (1-f) + \mathbf{B}\ f. \tag{2.1}$$

Cosine interpolation provides much smoother result than linear interpolation. It is clearly better if a slight loss of computation speed is not an issue.

Definition 2.2 Given points $\mathbf{A}$, $\mathbf{B}$ and the fraction f of the interval between them, $f \in [0,1]$, cosine interpolation $\mathbf{I}_{COS}$ between $\mathbf{A}$ and $\mathbf{B}$ at position f can be defined as

$$\mathbf{I}_{COS}(\mathbf{A},\mathbf{B},f) = \mathbf{I}_{LIN}(\mathbf{A},\mathbf{B},\frac{1}{2}\ (1-\cos(f\ \pi))). \tag{2.2}$$

Cubic interpolation gives very smooth result because the function respects gradients in both border points. However, low speed of such algorithm is a considerable disadvantage. The quality of the result might finally not be adequate the computation time. The algorithm is based on the same principle as cubic Bézier curves. The interpolation is performed on four points (instead of two border points as previous methods) – the border points **A** and **B**, the predecessor of **A** and the successor of **B**. These points form four control points of a cubic Bézier curve. The fraction f of the interval between **A** and **B** is adjusted to define that portion of the curve.

Definition 2.3 Given points $\mathbf{A}_{PRED}$, $\mathbf{A}$, $\mathbf{B}$ and $\mathbf{B}_{SUCC}$, which represent the predecessor point, both points of interpolation and the successor point, and the fraction f of the interval between $\mathbf{A}$ and $\mathbf{B}$, $f \in [0,1]$, cubic interpolation $\mathbf{I}_{CUB}$ between $\mathbf{A}$ and $\mathbf{B}$ at position f can be defined as

$$\begin{aligned}
P &= (\mathbf{B}_{SUCC} - \mathbf{B}) - (\mathbf{A}_{PRED} - \mathbf{A}) && (2.3)\\
Q &= (\mathbf{A}_{PRED} - \mathbf{A}) - P && (2.4)\\
R &= \mathbf{B} - \mathbf{A}_{PRED} && (2.5)\\
S &= \mathbf{A} && (2.6)\\
\mathbf{I}_{CUB}(\mathbf{A}, \mathbf{B}, f) &= P\,f^3 + Q\,f^2 + R\,f + S. && (2.7)
\end{aligned}$$

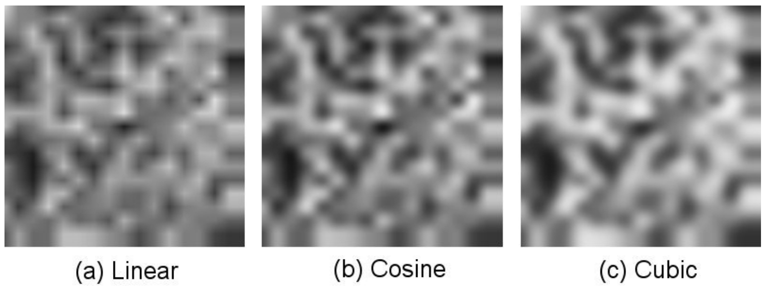


Figure 2.7: An example of two-dimensional interpolations.

Multiple layers

To produce natural-looking output, several noise functions of different frequencies are added together. It is recommended that the frequency of each noise function is double of the previous, thus $f = 2^i$ where f is the frequency and i is the i-th noise function being added. Two terms, *persistence* and *number of octaves* are used to control noise functions and their addition.

- **Persistence** is a value specifying the amplitude of the noise wave of given frequency.

  Definition 2.4 Given the frequency f of a layer of Perlin noise, in a form that $f = 2^i$, and persistence $\mathbf{P}$, the amplitude $\mathbf{A}(f)$ of noise function can be defined as

$$\mathbf{A}(f) = \mathbf{P}^i. \qquad (2.8)$$

- **Number of octaves** represents the number of noise functions being added, in other words the number of layers of Perlin noise.

Hardware implementation

Let's recapitulate the idea of Perlin noise. It consists of layers of smooth noise data. That is achieved by interpolating random noise values. Finally, layers are added together. See figure 2.6 for an example. All these steps can take considerable advantage of the computational power of modern GPUs. Nowadays, basically every graphic card on the market offers texturing with linear filtering. That is a very well suited instrument for generating smooth noise waves. Furthermore, blending capacity of the graphic hardware can be used to add noise waves together.

The hardware implementation brings several differences. The produced Perlin noise and it's each layer (each noise wave) is represented as a two-dimensional texture. That is because the whole operation is done into a viewport or an off-screen buffer, which are naturally two-dimensional. That means this approach can be used to generate sampled Perlin noise, but not a continuous function. However, for the reason of performance, it is desirable to hold the produced Perlin noise in a buffer instead of calculating every time it is needed.

At first, a noise texture is generated, using a common random value generator. This texture is used for generating interpolated samples, which are drawn one on top of the other using the blending capacity of the graphic hardware. Furthermore, this method can benefit from the ability of the hardware to use several texturing units in a single pass, so called *multitexturing*. Blending layers using various opacity factors equals to adding noise waves of various amplitudes, therefore it is possible to use the same noise texture for all the samples. The trick to the increasing frequency of added samples is using portions of the noise texture. Smaller portion of the texture represents a noise wave of lower frequency, on the other hand, the size of the whole texture represents the highest frequency available without repeating the noise.

There are several approaches to produce interpolated noise samples, based on the capacity of the graphic hardware and the quality of the output. Producing linear-interpolated samples is relatively simple and does not require specific graphic hardware and OpenGL extensions. A textured polygon is rendered to fill the whole viewport or off-screen buffer. If the portion of the texture mapped on the polygon is smaller than the viewport size and linear filtering is enabled on the graphic device, the rendering is naturally linear interpolated when the part of the texture is stretched to fill the desired space. This approach is very efficient, making it possible to produce Perlin noise on common graphic hardware in real time. For better interpolation methods, convolution filters can be used. However, this approach require more sophisticated graphic equipment with support for the *ARB_imaging* extension. The quality of linear-interpolated Perlin noise, as an easy, compatible and efficient approach, is shown in figure 2.8. This method gives satisfying result not much worse than cubic-interpolated Perlin noise, used as a reference quality measure.

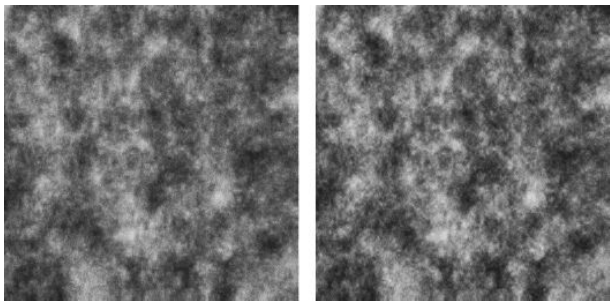


Figure 2.8: A quality experiment of five layer Perlin noise . Linear-interpolated Perlin noise (left) in comparison to the original approach, Cubic-interpolated Perlin noise(right).

2.5 The principle of rendering and animating

To bring the scene to life is basically the most important part of the whole process of making an intro. At the final state, the animation should be smooth and pleasant to watch. Furthermore, sound and music is integrated to emphasize the overall mood. A high level of synchronisation is desired.

The whole animation is commonly splitted into smaller logical portions (scenes). Finding appropriate algorithms and approaches to control the scene animation is essential for later synchronisation process. Using parametric algorithms is a well suited approach. In principle, the properties of objects involved in the scene are bound to variables, usually in the real numbers domain, which are adjusted by a control algorithm. The control algorithm, given a reference value, provides the value of the bound variable. An apparent and convenient way is animating the scene based on the time flow of the animation. A value of each variable controlling the animation is computed from the time interval.

A common and efficient approach is to employ polynomial functions in algorithms controlling the animation. Namely, Bézier curves and Béezier splines are easily and intuitivelly used. As being parametric curves, they are evaluated giving the specified time as a parameter. Additionally, nonuniform Bézier splines handle nonhomogeneous distribution of the time interval over the curve. That is convenient to control the acceleration produced on the curve.

2.6 First step of compression

In order to minimize the size of structures holding data for the rendering system, we introduce a simple model of data compression. This model is based on quantization of homogeneous arrays of values to minimize the bit-depth needed to represent each value. Note that this approach represents a lossy data compression. Generally, the situation is to express real numbers with floating-point precision as numbers with fixed-point precision using less bits. Furthermore, we propose a refinement of common quantization approach to adapt to the internal structure of data used in the rendering system.

Let's put an example of data structure holding control points of Bézier surface, which is intended to be widely used. The surface consists of several patches. Each patch is defined by 16 control points, each of a three-component vector of real numbers. Considering that each adjacent patch shares 4 edge control points and there is N patches in the surface, there is exactly $16 + (12 * (N - 1))$ control points in the surface. For a surface of 10 patches it is 140 control points resulting in a buffer of 1680 bytes (3 components per control point, 32 bits per a floating-point real number). In a real situation, the number of patches is considerably higher, resulting in several hundreds of control points in one surface. That makes Bézier surfaces significantly expensive on the size of input data.

Quantization

Let's look at the data to be compressed. It will be primarily arrays of control points, normals and texture coordinates. Generally, it is not possible to predict the distribution of the data over its bounding area. Therefore, an uniform quantization is preferred and, in real situations, it provides good results. See fig. 2.9 for a test of quality of the produced data.

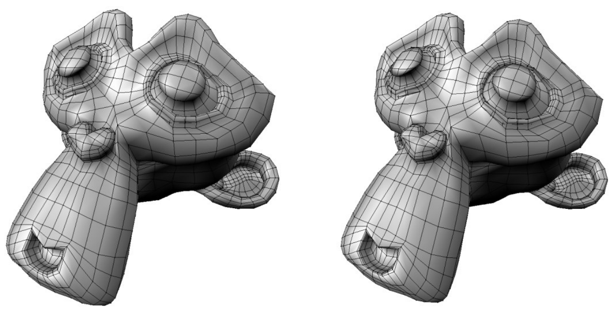


Figure 2.9: A comparison of quantize-coded mesh (right) with the original (left). A 32 bit floating-point vertex buffers were quantized to 10 bit fixed-point precision, a 32 bit floating-point buffer of normals was quantized to 8 bit fixed-point precision. The mesh consists of 1980 vertices and the same amount of normals, resulting in 47 520 bytes of data(the original approach) and 13 365 bytes of quantized data. The subjective visual quality has been well preserved. The model has been taken from Blender standard models.

Interleaved arrays

Eventual enhancement of the coding technique comes from further investigation of the data to be compressed. In important cases, such as arrays of vertices, there is usually an internal structure in the array itself. Let's demonstrate on an array of vertices. It is a sequence of three-component vectors representing each vertex in three-dimensional space, eg. it consists of values for X, Y and Z axis. See fig. 2.10. It is, then, possible to quantize each component separately, using various bit depths. This approach clearly benefits from the knowledge of the structure topology to reflect the bit-depth precision used to quantize each component. In case of an irregularly sized object, for example, the use of various bit depths for each axis (eg. X, Y or Z) allows more precise coding maintaining the same number of bits.

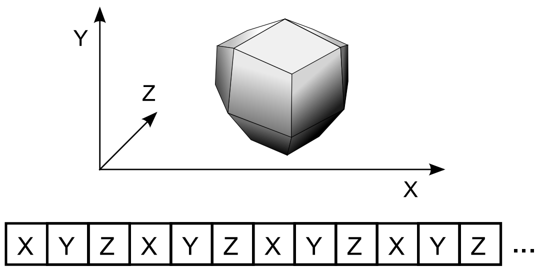


Figure 2.10: A structure of an interleaved array of vertices. Each vertex is a three-component vector representing values in X, Y and Z axes respectively.

Chapter 3

DemoBasic library

The main challenge in making a demo is clearly it's limited size. Apart from expressing ideas or telling a story, there certainly is a satisfaction for programmers to show *how much can be packed in such a small amount of code*. Despite of it's small size, the animation should appear complete, producing comfortable and enjoyable experience for a viewer. Most of todays intros are developed for Microsoft Windows for it's uniformity and good support of modern graphic hardware and tools.

One way to decreasing the size of the output executable code is clearly the right choice of a compiler. Yet more important, there are techniques to minimize the size after compilation using compressing techniques.

3.1 Programming language and compiler

Earliest demos were typically made in machine code monitors, the same programs that were used by the crackers to crack copy protections. The next step was the transition from monitors to assemblers. Higher-level programming languages, such as C and C++, started to gradually take over assembly programming in the demos of the 1990s, when cycle-level timing was no longer considered as important as before and compilers were beginning to be able to produce code comparable to hand-coded assembly. The transition to higher-level languages originated in the PC scene. Nowadays, demos programmed in pure assembly are rare on the PC, except for the extreme size-restricted categories.

C and C++ are well suited for making size restricted intros, as they provide enough flexibility and compilers produce native binary code. There is a wide variety of C anc C++ compilers for Microsoft Windows, some are more suitable than others though.

- **MinGW – Minimalist GNU for Windows** is a collection of freely available and freely distributable Windows specific header files and import libraries combined with GNU toolsets that allow one to produce native Windows programs that do not rely on any 3rd-party C runtime DLLs.

- **Microsoft Visual Studio** is a comercial solution which provides a powerful tool for making native Windows applications. It contains highly optimized C and C++ compiler. However, the produced program relies on a third party runtime DLL, which is not present in all configurations of Microsoft Windows.

- **Intel C/C++ Compiler** offers excellent compiler and other tools for developing and tuning Windows applications which do not rely on any 3rd-patry runtime libraries. It is distributed for comercial purposes though and is available to buy or limied trial only.

A question whether to use pure C or more flexible C++ came up. C++ offers a fancy and well arranged coding, as well as further optimizations. However, it is necessary to consider the size of the output code. Because no really reliable investigation was found, a simple experiment took place to answer this question. A basically random code, yet functional and relevant to computer graphics, OpenGL and this project, written in pure C was gathered from several sources, put together and compiled. Furthermore, the same code was gradually transformed into C++, using namespaces, classes, user-defined operators, templates and memory allocation using the *new* operator. MinGW – Minimalist GNU for Windows was used for compiling, as it was also the decision for compiling the whole project, using *gcc* as the compiler. Also, the Ultimate Packer for eXecutables (UPX), see section 3.2, was used to see the final output size, as it was also the decision for reducing the size of the final executable in this project.

Compiling the C code, removing unused resources, as debugging symbols, and compressing produced a 24Kb executable. The transformed code using namespaces for better formatting of the code and defined operators for vector algebra produced, after removing unneccessary resources and compressing, a code of the same size. A slightly bigger code was generated when classes with defined constructors were involved. The difference was approximately 2Kb. However, the use of templates and the *new* operator required the extended compiler, *g++*, for compilation. The difference in code produced by *g++* was approximatelly 10Kb. For that reason it was decided not to use templates, C++ memory allocation and further complex C++ constructions.

3.2 Compression

Executable compression has been used in demos since the very beginning. Pirated software needed to be packed into a compact and easily spreadable format, which often required some kind of compression for both the software itself and the attached intro. Early demos often had multiple parts which were separately decompressed into memory during the short pauses between parts. Executable compression is any means of compressing an executable file and combining the compressed data with the decompression code it needs into a single executable. There are compressors designed specially for small intros, however demogroups usually do not publish their compression tricks in order to keep the advantage. There are several common executable packing programs though.

- **Ultimate Packer for eXecutables (UPX)** is an open source executable packer supporting a number of file formats. It is free software, released under the GNU General Public License. It achieves an compression ratio of approximatelly 40-70%, depending on the source program, and offers very fast decompression. The executable suffer no memory overhead or other drawbacks because of in-place decompression. See table 3.1 for details. UPX uses a lossless compression algorithm called UCL, which is a free implementation of the proprietary NRV, *Not Really Vanished*, algorithm. UCL requires no additional memory to be allocated for decompression, a considerable advantage that means that a UPX packed executable requires no additional memory.

- **PECompact2** by Bitsum Technologies is a next generation win32 executable/module compressor. It provides slightly better compression than UPX compressor (see table 3.1). It is licenced for comercial use and is only available to buy or as a limited trial.
- **ASPack** is an advanced Win32 executable file compressor, capable of reducing the file size of 32-bit Windows programs by as much as 70%. It is licenced for comercial use and is only available to buy or as a limited trial. See table 3.1 for details.

Compressor	120 320 bytes source	43 064 bytes source	**COMPRESSION RATIO**(%)
UPX	61 440 bytes	21 504 bytes	48.94 / 53.06
PECompact2	53 760 bytes	19 417 bytes	55.04 / 54.9
ASPack	66 560 bytes	24 064 bytes	44.69 / 44.12

Table 3.1: Compression quality experiment.

There are few drawbacks using compressed executable files. Some (usually older) antivirus software reports compressed executables as viruses. Compressed executables have a greater impact on system resources. The operating system cannot read their decompressed images on demand from the disk, like it would with normal, uncompressed executables. Instead, the decompressor usually allocates a block of memory to hold the decompressed data, which stays allocated as long as the executable stays loaded, whether it is used or not. However, considering the size of the animation in tens of kilobytes, it is not an issue on today's computers.

3.3 Design

The goal was to design and implement a suitable library which could be used for future animation making. Size of the produced executable code was the main criteria. Furthermore, the library is intended to be transparent and easy to use. It is written in C++ and organized into namespaces, each covering a specific functionality. See fig. 3.7 for a view on namespaces present in the library.

Initialization

Each namespace contains initialization function *init()* which does not take any argument and returns *true* on success and *false* otherwise. The purpose is to allow to set-up the necessary environment, often including pre-calculation of commonly used data, retrieving pointers to OpenGL functions and others.

Query

Each namespace contains functions to query data relevant to its functionality. Principally, it is data retrieved by previous initialization function call. Currently, there is an integer-returning version only.

```
int geti(int enum_what);
```

Section 3.4 describes specific aspects of each namespace.

3.4 Implementation

DemoBasic library, in general, implements and automates techniques and constructions frequently used in computer graphics. Please see the referenced literature, namely the book *Moderní počítačová grafika* [1], for an overview on this topic. Further information can be also found on Gamedev.net [7], a website devoted to interactive computer graphics. In the following sections is an overview of functionality implemented in DemoBasic library with a description of selected tasks specific to intro-making.

3.4.1 Video

Namespace *video* provides automation to Windows API *(WinAPI)* calls related to opening a window and further tasks on its device context. It offers the minimum necessary functionality to OpenGL rendering under Microsoft Windows.

Initialization

Function *create()* opens a window with WS_OVERLAPPEDWINDOW style and WS_EX_APPWINDOW extended style. Naturally, it requests a pixel format descriptor supporting OpenGL, drawing to window and double-buffering. Furthermore, for better image quality, graphic device is inspected for presence of a pixel format supporting multisample anti-aliasing *(MSAA)* and stereographic rendering.

```
bool video::create(char *title, int width, int height, int bits, bool fullscreen);
```

Its initialization function covers setting-up initial OpenGL state and querying useful information on implementation specific OpenGL capabilities. Following information is retrieved and available for further use to the application.

```
int video::geti(int enum_what);
```

enum_what:

N_MAX_LIGHTS	the number of light sources
N_MAX_VIEWPORT_SIZE	maximum supported viewport size
N_MAX_TEXTURE_SIZE	maximum supported texture size
N_MAX_LIST_NEST	maximum display list nesting level
N_ALPHA_BITPLANES	the number of alpha bit planes present in the pixel format
N_DEPTH_BITPLANES	the number of depth buffer bit planes
N_ACCUM_RED_BITPLANES	the number of accumulation buffer red bit planes
N_ACCUM_GREEN_BITPLANES	green bit planes
N_ACCUM_BLUE_BITPLANES	blue bit planes
N_ACCUM_ALPHA_BITPLANES	alpha bit planes
N_MULTISAMPLE	multisample anti-aliasing support
N_STEREO	the presence of stereographic buffers

Device context

The functionality of this namespace lays in providing rendering canvas for OpenGL. Before each frame, the device context should be made current with *select()* function. After, buffers are swapped with *swapBuffers()* function.

```
void video::select();
void video::swapBuffers();
```

Multisample anti-aliasing

Anti-aliasing techniques has been developed since the very beginning of computer graphics in order to minimize the distortion artifacts appearing after image rasterisation. It is usually done by supersampling, an anti-aliasing technique of rendering the image at higher resolution and downsampling. Multisample anti-aliasing generally refers to a hardware implementation of supersampling technique with several optimizations. For further information please refer to [1].

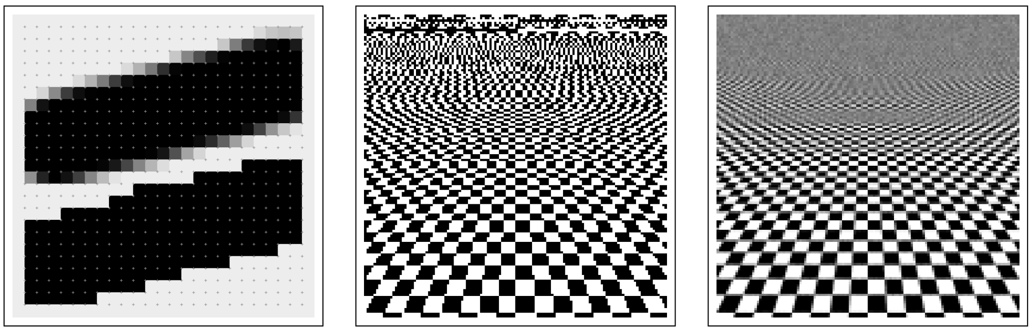


Figure 3.1: Anti-aliasing improving image quality. An aliased and anti-aliased line (left). A perspective projected view on a checkerboard with no anti-aliasing (center) compared to anti-aliased (right). Checkerboard images taken from Wikipedia, The Free Encyclopedia [12].

Employing MSAA is a little tricky. It requires an additional step to query for presence of a pixel format supporting MSAA. It is OpenGL specific task, in the BasicLibrary is implemented through *ARB_multisample* extension. However, we cannot request the presence of ARB_multisample extension and the MSAA support unless there is a valid device context. The gist is to open a window with common pixel format supporting OpenGL. With this widow being set as the current device context, we can query a the presence of ARB_multisample extension and support of appropriate pixel format. Eventually, we renew the window and update the pixel format to reflect the result of the query. Figure 3.2 shows briefly the process of getting pixel format and device context supporting MSAA.

Stereographic rendering

Stereography is a visualization technique creating the illusion of depth in an image by providing a slightly different image for each eye. It dates back to the mid 19th century

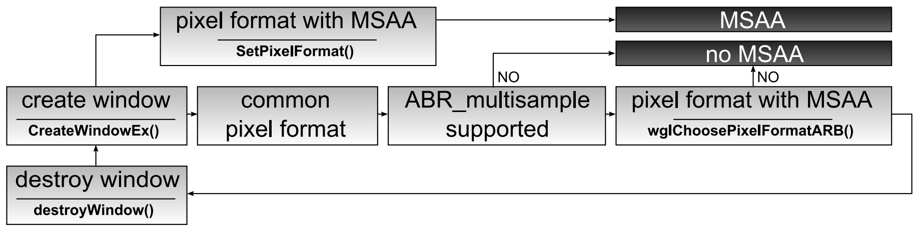


Figure 3.2: A brief diagram of multisample anti-aliasing initialized in WinAPI.

when Sir Charles Wheatstone, a british scientist and inventor, described the principle of human perception of depth. He showed that the impression of solidity comes from combining two separate images taken by both eyes from slightly different points of view. He produced several stereographic drawings and constructed stereograph (also called stereoscope), an instrument designed to watch stereographic cards. The construction has been improved and considerably lightened over the time, however, the principle remains.

Figure 3.3: An old Zeiss pocket stereoscope with original test image. Image taken from Wikipedia, The Free Encyklopedia [12].

Stereographic rendering involves an additional framebuffer for each rendering target to hold rendered frame for each eye. WinAPI pixel format descriptor natively covers stereo imaging capacity. Therefore, one just specifies stereo buffers request in the pixel format descriptor. If enabled, OpenGL will support left and right version of each front (and back) framebuffer.

Stereoscopic buffer is requested:

```
PIXELFORMATDESCRIPTOR pfd;
...
pdf.dwFlags |= PFD_STEREO;
```

```
...
SetPixelFormat(...);
```

The currently set pixel format can be queried whether various flags has remained set or not. If the user's graphics configuration is not set up for stereo buffering, Windows will not allow the PFD_STEREO flag to be set.

```
int PFI = GetPixelFormat(hDC);
PIXELFORMATDESCRIPTOR pfd;
DescribePixelFormat (hDC, PFI, sizeof(PIXELFORMATDESCRIPTOR), &pfd);
if ((pfd.dwFlags & PFD_STEREO))
...
```

Stereographic rendering requires a slight intervention to the vertex transformation process. The scene is rendered separately for the left and right buffer. An appropriate view displacement is applied to each projection matrix for left and right eye rendering. The projection matrices can be pre-computed with *calcStereoPerspective()* function.

```
void video::calcStereoPerspective(float parallax_angle,
                                  float FOV,
                                  float focal_length,
                                  float separation);
```

`parallax_angle`	angle of displacement of the left and right eye image recommended to be kept less than or around 3 % of viewport width
`FOV`	projection filed of view
`focal_length`	focal length, thus, the frame of focus
`separation`	separation of the two images in horizontal axis influences the strength of dept illusion produced

Stereographic rendering is an experimental feature of DemoBasic library and is not included in the final animation.

3.4.2 Time

In order to make the animation flow consistent, an internal time counter is implemented. It is being synchronized with system time.

Initialization

This namespace currently does not need any initialization. The timer is started explicitly using *start()* function. That allows to begin the time flow from a specified point of program, eg. after all data is generated and the anomation begins to play.

```
void time::start();
```

Functionality

Internal time value can be queried with *getLocal()* function returning the time in seconds.

```
float time::getLocal();
```

Furthermore, the time flow can be paused and resumed with *pause()* and *resume()* functions.

```
void time::pause();
void time::resume();
```

3.4.3 Images

DemoBasic library is able to hold three types of images.

- simple raster images in *sImage* and *sImagef* structure for unsigned char and float data respectively
- RLE encoded raster images in *sRLEImage* structure
- vector images in *sVectorImage* structure

3.4.4 Texture

Namespace *texture* is intended to automate work with OpenGL textures. Currently, only two-dimensional textures are handled. The automation includes creating an empty texture or a copy of user-defined raster data, setting filtering modes and procedural generation.

Initialization

This namespace currently does not need any initialization and does not have any attribute to query.

Functionality

Function *create2D()* creates a two-dimensional texture, either empty or as a copy of user-specified data.

```
unsigned int texture::create2D(sImage *image);
```

Furthermore, there are routines for procedural generation of textures. It is, namely, turbulence noise, pattern, linear and radial gradient and normal map generator. They have been designed and implemented to use the computational power of graphic device, thus, making the generation process significantly faster. Compatibility issues have been kept in mind also. Here we describe the implementation aspects of these techniques. For detailed information on the usage please see DemoBasic reference manual.

Turbulence

Function *turbulence2D()* generates a two-dimensional turbulence noise texture. It originates in Perlin noise, however, compared to the original approach, it is significantly simplified and lightened. Refer to section 2.4.1 for further information on the principle of Perlin noise.

There are two conceptual approaches. Firstly, the generation can be written whole as a fragment program, rendering the noise in a single pass. That would allow to employ real-time generated turbulence noise which might be, indeed, useful for various effects. An alternative is to use fixed-function rendering using OpenGL blending capabilities. The last method was chosen for the fact that we, at this point, do not intend to use real-time generated turbulence. Let's look how it is done. Primarily, we create an uniform noise texture, which will stand as our noise function.

```
sImage *image = noise::noise2D(type, size);
unsigned int T = texture::create2D(image);
```

To imitate the use of noise functions of various frequencies, we use various portions of this texture stretched over the entire output texture area. Filtering mechanism of the graphic device will do linear interpolation for us. Furthermore, the blending capacity will accumulate layers of the noise. Therefore, the whole generation comes in few OpenGL calls.

```
glBlendFunc(GL_SRC_ALPHA, GL_ONE);

for (int k = 0; k < detail; k++) {
    float texcoord_select[2] = {(float)(k) / (detail), (float)(k+1) / detail};
    glColor4f(color.x, color.y, color.z, (1.0 - ((k / (float)detail) )) * A);
    fillFrameBuffer(vec2f(texcoord_select[0]), vec2f(texcoord_select[1]));

}
```

We use $(sourcecolor) * (sourcealpha) + (destinationcolor) * 1.0$ blending equation. That makes the current layer scaled to desired amplitude and added to an off-screen buffer. A is blending opacity ratio, it controls the saturation properties of subsequent addition of layers.

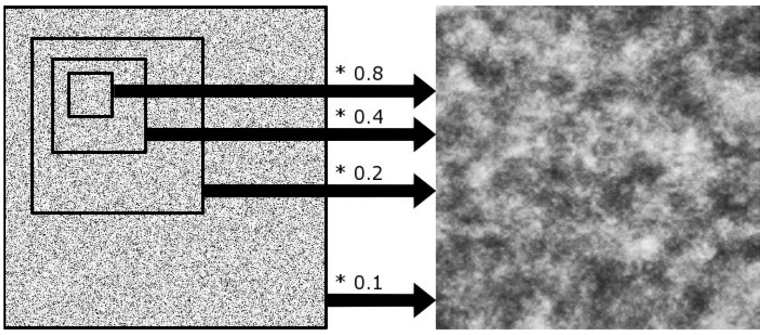


Figure 3.4: A principle of turbulence noise generation implemented in DemoBasic.

This method is easily optimized using multiple texture units, allowing to render the turbulence in significantly less rendering passes. Eventually, this approach can be used for real-time generation.

Pattern

Various textures such as woods, marbles or clouds can be achieved with *pattern()* function. The idea is to somehow distort the contents of an image, preferably with the data stored in another image. It is implemented using OpenGL fixed-function rendering. The pattern is rendered into a framebuffer or an off-screen buffer pixel by pixel with distorted texture coordinates for each. Filtering capabilities of the graphic hardware will do bi-linear sample filtering for us, significantly improving the output quality.

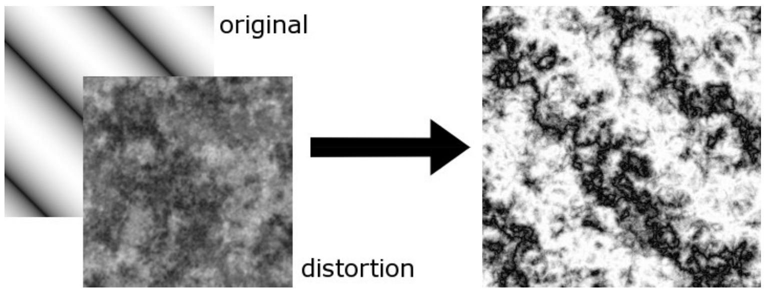


Figure 3.5: A principle of pattern generation implemented in DemoBasic. An original sine gradient texture is distorted with a turbulence. Various structures and materials can be imitated this way.

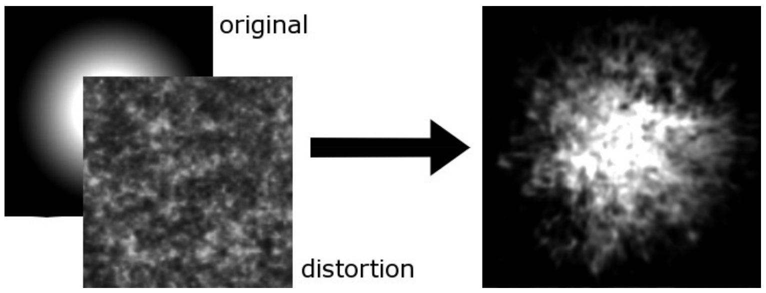


Figure 3.6: A principle of pattern generation implemented in DemoBasic. A radial gradient texture is distorted with turbulence texture to yield a cloud texture.

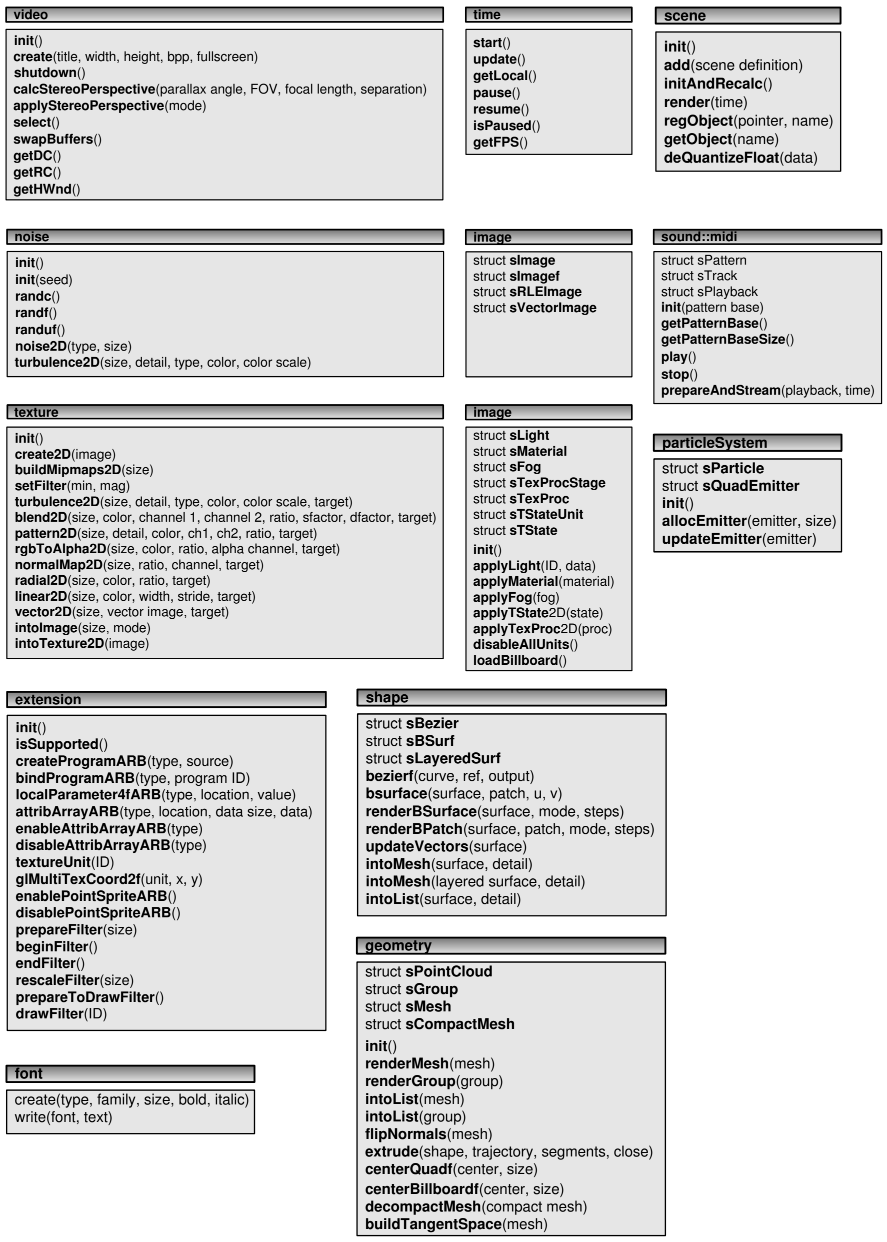


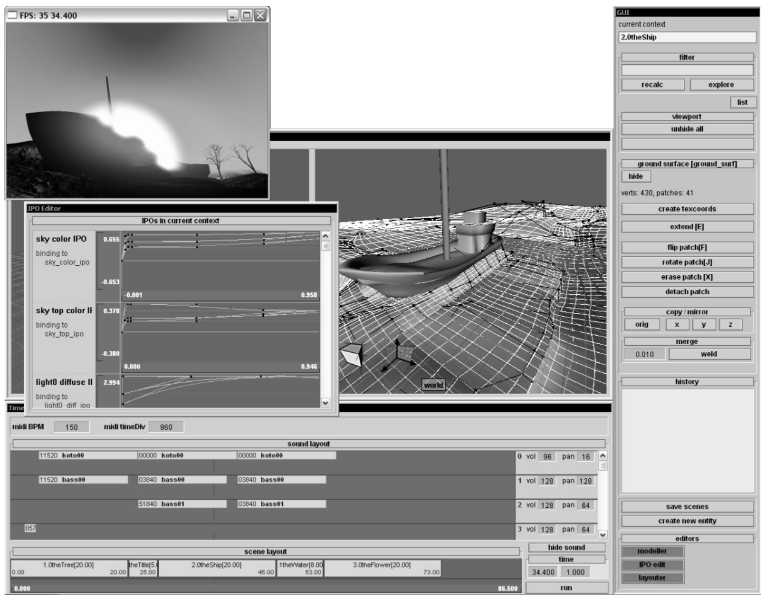
Figure 3.7: Namespaces in DemoBasic.

Chapter 4

DemoTool

DemoTool is a demo authoring tool providing convenient and intuitive approach to adjusting values of the scene design and animation. There is a range of these tools already developed by various demogroups and given for free-use to the public. Some of them are advanced all-in-one editors able to output the final executable code of the animation, others are rather simple tools to help with this or that in the animation making.

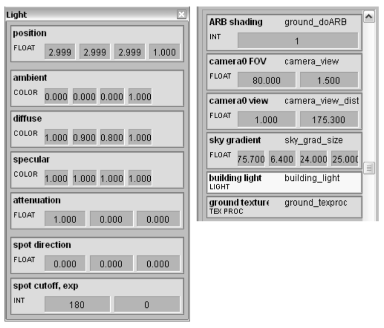
4.1 User interface



In the early stage of development, DemoTool was designed as a simple tool providing a friendly view on the internal data of the animation only. Eventually, it became a major part of the project and evolved into a complex set of tools with a graphic user interface. The aim was to allow interactive editing straight in the animation itself, thus, let the user view and change various properties while the animation is running. The demo tool is designed as a static library to be linked to the animation. Once the animation is finished, this tool is removed completely.

The user interface is divided into several parts. Firstly, it is an area of selection of defined contexts (scenes). For each context there is a list of currently bound variables. Generic arrays of data types (such as an array of floats) can be edited directly, working on other structures (as lights or materials) employs appropriate dialog boxes for intuitive editing. Furthermore, each scene can be arranged and animated using an in-place *Modeller* and *IPO editor*. Finally, the overall layout including scene cuts and sound is tailored using the *Layouter* tool. Read the section 4.3 for detailed description of these tools. Each tool operates directly on data of the animation, allowing to see the result in a real time. Following sections describe this mechanism.

4.2 Connecting to the DemoBasic library



```
Light
position        FLOAT   2.999  2.999  2.999  1.000
ambient         COLOR   0.000  0.000  0.000  1.000
diffuse         COLOR   1.000  0.900  0.800  1.000
specular        COLOR   1.000  1.000  1.000  1.000
attenuation     FLOAT   1.000  0.000  0.000
spot direction  FLOAT   0.000  0.000  0.000
spot cutoff, exp INT    180    0
```

```
ARB shading      ground_doARB
INT              1
camera0 FOV      camera_view
FLOAT            80.000   1.500
camera0 view     camera_view_dist
FLOAT            1.000    175.300
sky gradient     sky_grad_size
FLOAT            75.700  6.400  24.000  25.000
building light   building_light
LIGHT
ground texture   ground_texproc
TEX PROC
```

```
float camera_view[2]       = {80.0, 1.5};
float camera_view_dist[2]  = {1.0, 175.3};
float sky_grad_size[4]     = {75.7, 6.4, 24.0, 25.0};
state::sLight building_light  =
                          { {2.999, 2.999, 2.999, 1.0},
                            {0.0, 0.0, 0.0, 1.0},
                            {1.0, 0.9, 0.8, 1.0},
                            {1.0, 1.0, 1.0, 1.0},
                            {1.0, 0.0, 0.0},
                            {0.0, 0.0, 0.0},
                            {180, 0} };
              ⋮

tools::addVar((void*)sky_grad_size, 4,
                   "sky_grad_size", "sky gradient");
tools::addVar((void*)&building_light, EVAR_LIGHT, 0
                   "building_light", "building light");
```

Before we speak about the DemoTool, let's recapitulate the essential facts about the DemoBasic library. It consists of structures designed to hold data of objects that appear in the animation. Furthermore, it implements functions that operates on this data. For more information about the DemoBasic library see chapter 3.

DemoTool is primarily designed as a mechanism that allows watching and editing values of these structures in an intuitive and user-friendly way. The whole mechanism is then integrated in the animation itself, therefore, while the animation runs, it is possible to see and change values interactively. Basically, there are two types of data structures in the DemoBasic library. Firstly, it is elementary structures, with fixed length, used for defining a light source, material properties and others. Secondly, there are variable-length complex structures used for example to hold data of a parametric curve or surface.

The data of the animation and of the rendering system is available to the DemoTool through a mechanism internally called *data binding*. The principle is to provide a pointer to the data structure defined in the animation code and an information on the type of the structure. The DemoTool then classifies the information and places appropriate controls in the graphic user interface to allow interactive and friendly editing. After editing, the data can be saved in a C source code syntax and included in the code of the animation.

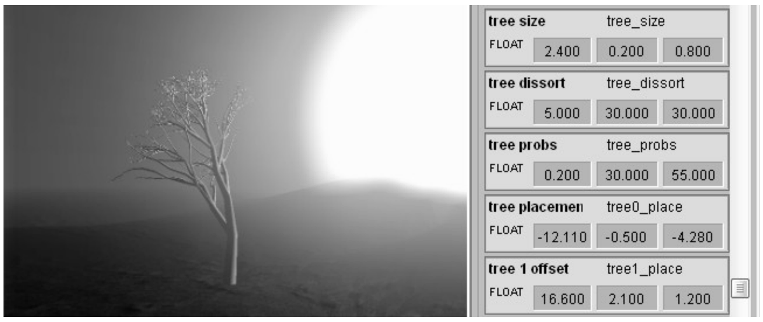
DemoTool is currently able to bind and provide interactive editing of the following.

- Generic homogeneous arrays of floating-point and integer data.
- Elementary structures of DemoBasic library including sLight, sMaterial and sFog.

- Procedural-generated texture definition.
- Texture state definition.
- Bézier curves and surfaces, meshes and point clouds in the Modeller tool.
- Interpolated values in the IPO editor.

4.2.1 Basic editing

The very basic functionality of DemoTool is to allow to watch end edit generic data arrays. Eventually, specific dialog boxes were employed for more complex data including light sources, materials or fog definition.



tree size	tree_size			
FLOAT	2.400	0.200	0.800	
tree dissort	tree_dissort			
FLOAT	5.000	30.000	30.000	
tree probs	tree_probs			
FLOAT	0.200	30.000	55.000	
tree placemen	tree0_place			
FLOAT	-12.110	-0.500	-4.280	
tree 1 offset	tree1_place			
FLOAT	16.600	2.100	1.200	

Figure 4.1: An example how various data are bound to help to control procedural generation of a tree.

4.2.2 Custom tools for complex data structures

Modeller

In the very beginning, modeller has been designed as a tool allowing to inspect the process of geometry procedural generation. Since Bézier surfaces became the main intended mean of representing geometry, it evolved into a modest modeling apparatus. Primarily, it is not intended to stand as a fully featured modeling and animating tool. It is proposed to import complex models from a third-party modeling solution. Modeller handles editing Bézier curves and surfaces, inspecting meshes and placing and editing light sources.

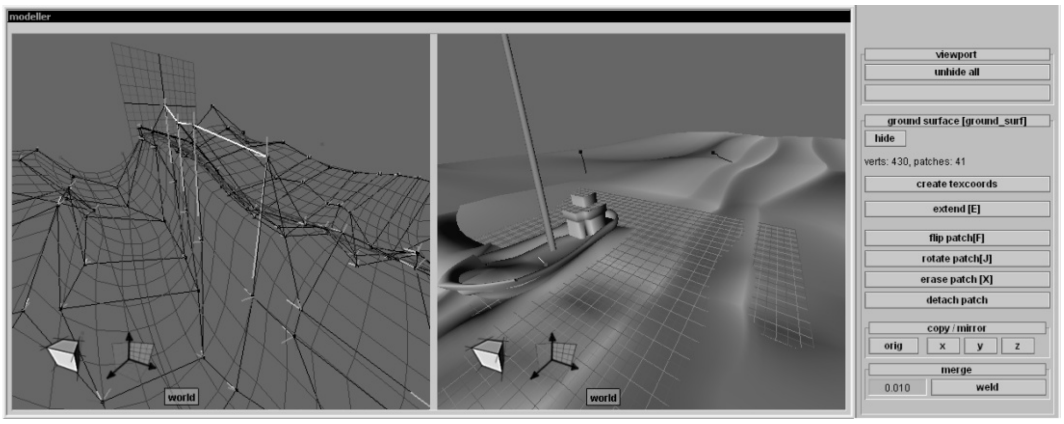


Figure 4.2: Modeller.

IPO editor

IPO editor has been developed in order to provide interactive control of variables which value is interpolated based upon the internal animation time.

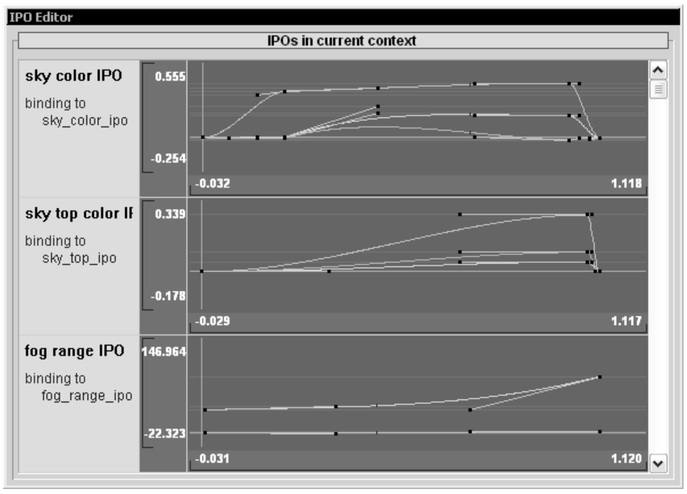
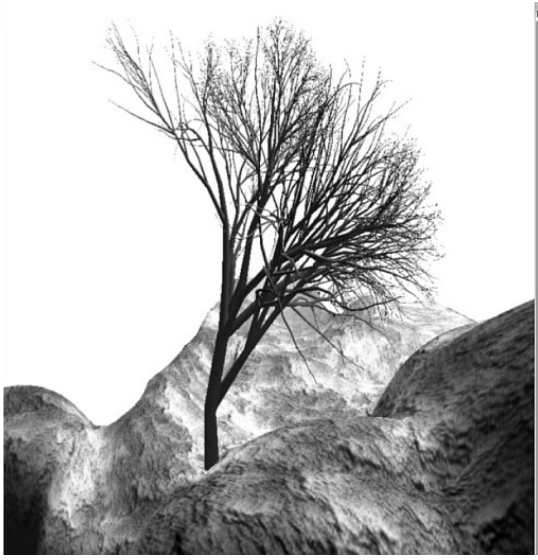
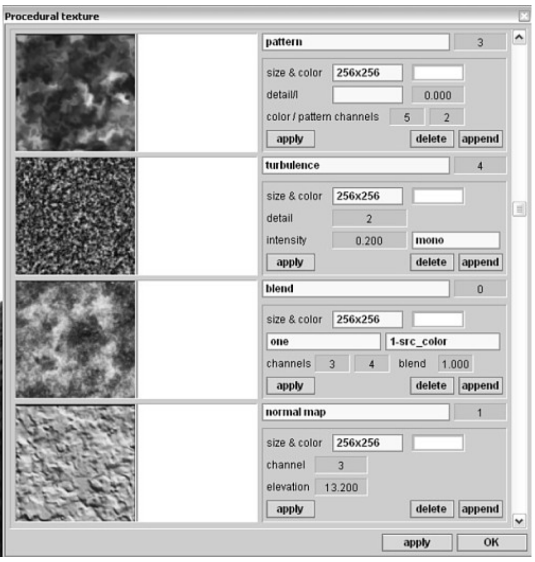


Figure 4.3: A view on IPO editor showing currently bound data. See how sky colors and fog range can be edited with cubic splines.

Texture-generation procedure

Texture procedural generation is realized through a sequence of operators. A specific tool helps us to edit the texture procedure. Preview is provided for each operator for more intuitive work.





Procedural texture

pattern 3
size & color 256x256
detail/l 0.000
color / pattern channels 5 2

turbulence 4
size & color 256x256
detail 2
intensity 0.200 mono

blend 0
size & color 256x256
one 1-src_color
channels 3 4 blend 1.000

normal map 1
size & color 256x256
channel 3
elevation 13.200

Figure 4.4: An example of proceduraly generated ground texture. A normal map is also generated for per-pixel operations.

4.3 Animating with DemoTool

DemoTool currently offers some simple animating mechanisms. Let's remind that it is primarily designed to help with intuitive editing animation data structures. The rendering loop is realized through a rendering function embedded in the animation source code. DemoTool does not provide any mean of automation to rendering functions. The reason for that is to keep maximum freedom for the animation itself. See section 2.5 on the principal concept of scenes and their rendering.

Chapter 5

Conclusion and future work

We studied the topic of short size restricted animations, finally resulting in a selection of techniques used. The major focus was put on constructing a functional base for further animation development. Eventually, two components came out. Firstly, it is DemoBasic library, which provides automation to selected graphic techniques and routines. The essential attributes of DemoBasic library were considered:

- **The smallest code size possible.** The library is written in C++, however, without complex constructions such as templates or classes. In contrary, namespaces and defined operators were used for better readability. That does not make the code size grow.
- **Good readability and ease of use.** The library is logically separated into namespaces, each covering specific functionality.
- **Automation of techniques frequently used in an intro.** The whole library is designed as an automation tool for various techniques of computer graphics or data coding. Then, we can employ complex mechanisms just with few lines of code.
- **Compatibility.** The library is designed to allow to use hot features of new graphic hardware. However, it provides fallback in case the current implementation does not support the requested functionality.
- **Performance.** Performance was not considered the vital aspect of the library. Nevertheless, methods were generally optimized. A significant loading time speed-up was accomplished.

The second product is DemoTool application, an visual authoring tool providing convenient and intuitive way to the making of the animation. Its graphic interface allows to watch and edit data structures bound from the animation code, inspect generated geometry or texture data, layout and animate scenes or synchronize with sound. The main features are:

- **Intuitive use**. Scene editing is definitelly better with mouse clicks rather than typing data directly to the structures of DemoBasic library.
- **Synchronization.** With DemoTool, we can insert MIDI patterns which will be eventually mixed together and played. A proper location in time can be fine-tuned

for each pattern to start exactly where we need. The same can be done with scenes present in the animation. This way, one can make the music and graphics play along.

- **All in real-time.** Changes to the animation data bound to DemoTool are reflected in real-time. Thus, we can immediately see how our change affect the animation.
- **Export the data.** Naturally, we wish to join the edited data with the animation again. DemoTool saves the data in C header files *(.h)* in a format of DemoBasic library structures.

5.1 Image quality

We tried to enhance the rendering quality and move the animation out of the bounds of common real-time rendered graphics. The image quality commonly suffers from jagged edges due to aliasing of neighboring samples. Also, we all know already the ordinary look of fixed-function pipeline rendering. The situation is, however, rapidly changing with the power of custom vertex and fragment programming. Custom per-pixel lighting, implemented with *ARB_vertex_program* and *ARB_fragment_program*, brings further detail and atmosphere to the rendering. The animation employs per-pixel lighting with bump-mapping for a single light source.

Furthermore, we apply various filters to enhance the final look. Firstly, it is *blur filter* for scenes under water, bringing a bit of the underwater feeling. A specially tailored form of blur is used for *glow filter* used to make selected objects appear as intensive light sources or very shiny.

5.2 Future work

At last we propose possible future work, improvements and additions. Firstly, DemoBasic library can be extended to implement other interesting mechanisms including skeletal animation, optionally with inverse kinematics. It is a powerful instrument for flexible geometry animation. The library is prepared to hold geometry with bones. Further proposal on enhancement of the geometry animation is to implement *Vertex key-framing.* The idea is to store multiple vertex arrays within one geometry object, each representing desired shape. Those can be eventually blended together with custom weights. It is a technique used for example in Blender for advanced animation such as facial expressions [6]. This technique is relatively simple to implement and optimize using vertex programs.

Also, some optimizations would be beneficiary. Namely, rendering of geometry can be done through *compiled vertex arrays* or *vertex buffer objects.* These techniques are relatively short to implement, however, are not supported on older hardware. Generally, we should be careful to implement the necessary functionality only.

Bibliography

[1] B. Beneš J. Žára, P. Felkel and J. Sochor. *Moderní počítačová grafika*. Computer Press, a.s., 2005. ISBN: 80-251-0454-0.

[2] Perlin K. Uses of integration in image synthesis. In *ACM SIGGRAPH Conference*, 1985.

[3] Perlin K. Improving noise. *Computer Graphics*, 35(3), 2002.

[4] WWW pages. Answers.com. `http://www.answers.com`. [Online; accessed 20-June-2007].

[5] WWW pages. All About OpenGL Extensions. `http://www.opengl.org/resources/features/OGLextensions/`. [Online; accessed 20-June-2007].

[6] WWW pages. Blender.org. `http://www.blender.org/`. [Online; accessed 20-June-2007].

[7] WWW pages. Gamedev.net. `http://www.gamedev.net/`. [Online; accessed 20-June-2007].

[8] WWW pages. OpenGL Overview. `http://www.opengl.org/about/overview/`. [Online; accessed 20-June-2007].

[9] WWW pages. SGI - OpenGL. `http://www.sgi.com/products/software/opengl/`. [Online; accessed 10-June-2007].

[10] WWW pages. SGI - OpenGL Extension Registry. `http://oss.sgi.com/projects/ogl-sample/registry/`. [Online; accessed 20-June-2007].

[11] WWW pages. the Khronos Group - Media Authoring and Acceleration. `http://www.khronos.org/`. [Online; accessed 20-June-2007].

[12] WWW pages. Wikipedia, the free encyklopedia. `http://www.wikipedia.org`. [Online; accessed 20-June-2007].

[13] Akeley K. Segal M. The OpenGL® Graphics System: A Specification. `http://www.opengl.org/documentation/specs/version2.0/glspec20.pdf`, 2004. [Online; accessed 10-June-2007].

[14] Neider J. Davis T. Shreiner D., Woo M. *OpenGL - Průvodce programátora*. Computer Press, a.s., 2006. ISBN: 80-251-1275-6.