Realtime Visualization Methods in the Demoscene

Boris Burger, Ondrej Paulovič, Miloš Hašan
burger@nextra.sk, twinsen@mayhem.sk, milos.hasan@inmail.sk

Department of Computer Graphics and Image Processing
Faculty of Mathematics, Physics and Informatics
Comenius University
Bratislava / Slovakia

Abstract

This paper introduces a concept of the "demoscene" phenomenon arising mainly among the lines of students who share interest in computer graphics and multimedia creations. The paper discusses the tight relationship between realtime visualization techniques and the subject of our focus which will be the so-called *demos* and *intros*. These are together the most attractive production type among the respective categories of creations the demoscene produces and also the ones most dependent on the methods introduced in the field of computer graphics – conceptual, mathematic or algorithmic. The authors further present their own work in this area with emphasis on describing some of the more interesting techniques they have invented or adapted for the sake of creating their own demos and intros.

KEYWORDS: realtime, rendering, demoscene

1 Introduction

The *demoscene* is a society of computer enthusiasts, arising since the mid eighties consisting mostly of university and high school students sharing active and usually productive interest in computer graphics, music and related multimedia.

The concept of *realtime visualization* [1] has been fundamental to the demoscene since its very beginning. It started with short one-man machine code programmed graphic effects on 8-bit computers pushing the hardware to the limit, to today's demos consisting of work by several people. The people involved in making a demo are *graphic artists*, together with *musicians* giving the demo it's own feeling and finally *programmers* utilizing plethora of visually attractive realtime techniques and algorithms to create various effects, putting the pieces of a demo together to create the work of art.

The *sceners*, which is a term the members of demoscene use to designate themselves, form into *demogroups*. As insinuated, demogroups consist of one or more

coders, graphic artists and musicians, typically from the same or close geographical location. With today's rise of Internet, which itself acts a significant role in the history and evolution of the *scene*, more and more often it happens that international groups with members from Finland, Poland, Norway or even Japan are formed. The other benefit of Internet lies in a fast and simple access to the demoscene production through archives maintained around the world [2].

For the sake of meeting and competition with other *sceners* in a friendly atmosphere there are so-called *demoparties* organized during the year in different parts of the world, mainly Europe. A *demoparty* is an annual event, being a gathering of people interested in the demoscene combined with competing in categories such as *demos* and *intros* – which are in fact demos with a size limit typically set to 64KB, 4KB or 256 bytes [3]. There are categories suited for musicians and graphicians to compete in *music*, *pixel graphics* (hand-drawn pixel by pixel), *raytraced graphics* and many other, depending on the respective party. The greatest demoparties held are *Assembly* [4] in Finland and *The Party* [5] in Denmark organized annually for already a decade. Another well-known party is *Mekka-Symposium* [6] that takes place in Germany. The number of visitors ranges from hundreds coming to smaller "local" parties to two or three thousands visiting international events like the ones mentioned previously.

The structure of our paper is as follows: sections 2 and 3 cover the history and present of the demoscene from a computer graphics and programming point of view. The rest of the article, presents three of the demoscene contributions authors of this article participated on, discussing three selected techniques utilized in their creation: a method for parametric representation of a rollercoaster track, creation of light effects using center projection of an object texture onto an environment and painting textures using implicit functions.

2 Evolution of demomaking

The roots of the demoscene date back to the era of 8-bit computers. Groups of *crackers* focusing on a removal of copy-protections from computer games used to attach a so-called *cracktro* in front of a game they *cracked*. It was a small realtime generated visual presentation of their group. The speed of 8-bit computers was very limited then, so these *cracktros* consisted often only of a scrolling text, some really simple bitmap effects or sometimes a picture in the background.

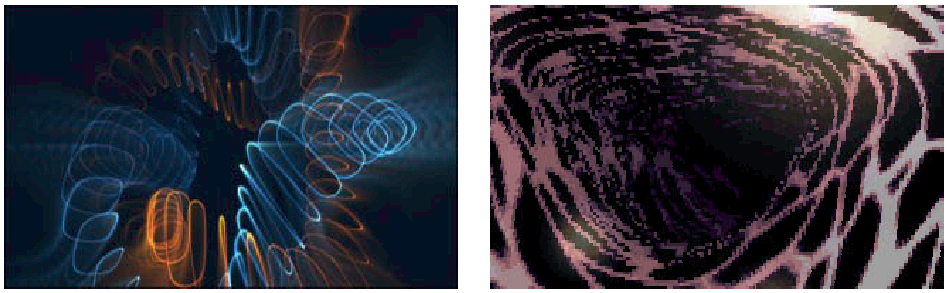
As the time went on, standalone *demos* started to appear – with their content resembling *cracktros*, but they were not bound to a game and were already completely departed from crackers' activities. In those *demos* authors were trying to push the limits of the current hardware creating new, original, and visually attractive effects. Speed was the reason why all *demos* were programmed using machine code back then. Effects were mostly 2D those days – scrollers, various bitmap distortions, plasmas and other. The most used computer among *sceners* was *Commodore* 64 [7], addressed to be the platform where the whole phenomenon began.

Figure 1: Examples of simple effects: *copper*, *fire*, *rotozoomer* and a *voxel landscape*. Screenshots taken from *Show* by *Majic* 12, *Inconexia* by *Iguana*, *Second Reality* by *Future Crew*, and *Airframe* 64KB intro by *Prime*.

Sometime around the year 1986 a new groundbreaking platform for demosceners appeared – *Commodore* AMIGA [8]. AMIGA offered hardware for realizing far more complicated algorithms than before. Notably, the artistic design became a strong element characteristic for this period.

Some time later during the era of AMIGA, a new *Intel* 80x86-based *PC* [9] accompanied with *Microsoft*'s operating system *MS-DOS* started to establish its position as a computer suitable for home computing. Finally it became the most common platform among demosceners despite a great problem it possessed – most graphic and sound hardware was not compatible with each other. Rather limited amount of memory restricted the demo's execution to a specific computer configuration at times and that didn't help to spread the demos and intros widely across the *scene* borders.

To be able to fit into tight limitations of that day's computer, demo programmers invented a noteworthy class of effects. These were realized by direct manipulation of graphics hardware, circumventing the operating system such as certain sorts of *scrolling* text and the so-called *copper* or *multicolor* effects. Very popular among the newly introduced techniques was *rotozoomer* where the computer screen was covered by a rotating and zooming bitmap, which was a starting point for polygon texturing routines utilized in 3D renderers later.



(a) (b)

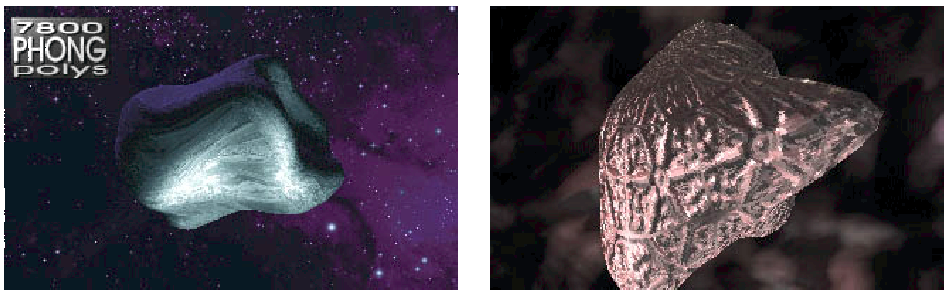
Figure 2: (a) A combination of 2D and 3D effects produces interesting results. Screenshot from demo *kkowboy* by demogroup *Blasphemy*, 1998. (b) Very original type of a tunnel effect, from demo *Xtal* by Finnish demogroup *Complex*, 1995.

The rapid evolvement of the PC platform naturally influenced the character of *demos* and *intros* produced. Simple 3D effects appeared, displaying mostly just a single rotating object. Depending on the hardware's speed only the vertices used

to be rendered at first, followed by edges, and only later filled polygon surfaces. An interesting class of 3D effects originated in rendering of landscapes, where the landscape's surface was typically visualized using column voxels. This is done by casting a ray for each column of the pixel buffer. It was a simple ancestor of realtime raytracing.

The increasing power of CPUs allowed for faster rendering of textured polygons which opened a whole new scale of possibilities on the field of realtime rendering. A variety of shading models used to be commonly implemented then: constant flat shading, gouraud shading and fake phong shading. The fake phong shading effect was achieved by taking advantage of a special texture combined with computing the texture coordinates from orientation of the normal vectors at the mesh's vertices.

With the end of 90's approaching more complex algorithms saw the light of realtime implementation in demos such as marching cubes, bump-mapping or dynamic shadows. In this period the low-level machine code used to be employed mainly for the most critical parts of a program in the first place. The rest was then programmed using a higher level language like *C* or rarely *Pascal*.



(a) (b)

Figure 3: (a) An example of fake *phong shading*, as presented in demo *Dope* by *Complex*, 1995. (b) Realtime *bump mapping*, from demo *Solstice* by *Valhalla*, 1996.

Even though the phong shaded, environment or bump mapped single object scenes were very nice for the moment, they became boring later on. Thanks to the rapid growth in processing power of CPUs, visualization of more complex 3D scenes was able to emerge. This is how the era of demos based on scene graph renderers started to rise.

For some time at the beginning of this new evolutionary step of demoscene, the center of all interest when creating a demo moved to write a fast, convincing and stable realtime polygon rasterizer for use within a scene graph renderer. With the low-end computer processors being still not that fast to make this trivial, programmers had to spend most of their time writing optimized shading, texturing, alpha-blending and other rasterization details in machine code to be able to render convincing 3D scenes at interactive framerates. That is the reason why programmers scarcely had the time to do something else to help making a good demo. Thus the demos produced at the beginning of this era were – though complex at inside – mostly boring fly-bys.

At first, the vast majority of sceners ignored the appearance of *Microsoft*'s new generation of *Windows* operating systems and continued writing demos and intros

(a) (b)

Figure 4: (a) *Cartoon rendering* in it's beginnings, screenshot from 64KB intro *Paper* by groups *Psychic Link* and *Acme*, 1996. (b) *Dynamic shadows*, as presented in *The Fulcrum* demo by the group *Matrix*, 1998. One of the few demos that employed bilinear filtering of textures within software renderer at surprisingly high framerate considering the power of low-end CPUs back then.

using the older *DOS* platform. Only later on, in part due to the introduction of low-end graphics adapters with the ability of 3D acceleration the *scene* started to develop demos for this operating system. Unfortunately, the quality did not improve at once and most demos were still only simple 3D scene viewers.

3 Present trends

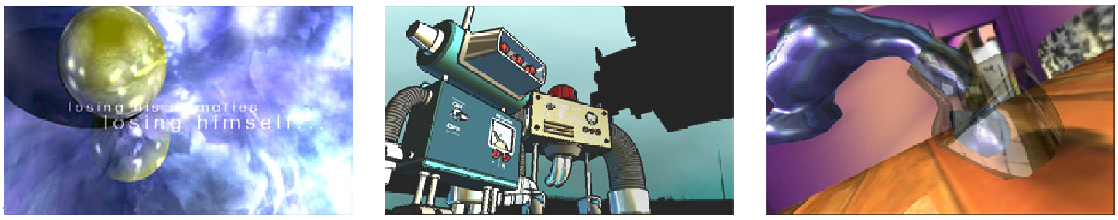
Demoscene is a subject to evolution since its early days. Even though the productive sceners still casually use assembler for its advantages, compose music in *trackers*, and draw graphics pixel by pixel, the mainstream is elsewhere.

The effects presented in demos are far more complex now. From simple texturing tricks and rendering single object scenes we have come to implicit surface polygonization [10], scene graph rendering, realtime raytracing [11] and radiosity, non-photorealistic rendering [12] and a plenty of other advanced techniques, like procedural generation of textures and geometry.

From a programmer's perspective, the code written is no longer so low-level than before. Concerning rendering and general visualization, the vast majority of production does not utilize any processor specific features and is typically written in a high level and portable language like *C++* [13].

With the introduction of sophisticated and fast graphics hardware – lately accessible to home computing – started the diversion from "classic" pure *software rasterization* methods to rendering *assisted by hardware acceleration*. Hence the demos of today do not depend so much on the underlying processor architecture, they rely on the graphics APIs (*Application Programming Interface*). There are two APIs widely accepted and used today: *OpenGL* [14] originated by SGI and *Microsoft*'s *DirectX* [15], the former being favorable for its elegance, overall design and platform, or to a certain extent, programming language indepedence. On the

other hand, *DirectX* gained many positive improvements lately and possesses a rich set of features, in part due to it's tight binding to the *Microsoft*'s platform.



(a) (b) (c)

Figure 5: (a) *Realtime raytracing* in 64KB intro *Heaven 7* by *Exceed*, 2000. (b) *Cartoon rendering* example from *Gerbera* by *Moppi Productions*, 2001. (c) *Scene graph rendering*, spherical environment mapping and implicit surface polygonization as seen in *Spot* by *Exceed*, 2000.

While the *Microsoft*'s *DOS* was the operating system of demoscene's choice for a long time period mainly due to its wide availability on low-end home computers, this is no longer true. The programmer's nightmare – either graphics or sound hardware incompatibility, was largely cured by the introduction of new generation operating systems suitable for home use. Though demoscene's mainstream uses *Microsoft*'s *Win32* platform, it is not uncommon to encounter demos bundled with binaries for multiple platforms including *Linux* or other Unix flavors. There is also an increasing number of demos produced for hardware architectures like *PowerPC*, *Alpha* and video game consoles.

Put together, the need for low-level programming fading away allowed the programmers to concentrate on the more challenging techniques and algorithms. Graphic artists and musicians take advantage of the new sophisticated software and so they are able to create more professional pictures, textures, and music score. At last, the *design* definitely finds its place in the PC demos and intros of today, being a very strong element.

4 Our work

In the following sections we present examples of our own work with emphasis on three selected techniques we have invented or adapted through the process of their creation.

4.1 DREAM demo

The *Dream* [16] demo was released in 1999 at the *Fiasko* demoparty held in *Uherské Hradiště*, Czech Republic. There it won the first place in the demo competition. It was a *DOS* executable, running also under *Microsoft Windows DOS* box. Unfortunately we have encountered certain compatibility problems causing that *Dream* does not run on every possible hardware configuration.

This demo featured several 2D effects and a scene graph renderer built on a top of a pure software rasterizer. The rasterizer employed *texturing* combined with *gouraud shading* using 8 bits of precision for each of the three color channels, supported by a *z-buffer* with 32 bits of precision. For the sake of fast rasterization we have utilized the *MMX* instruction set, which proved to be very useful when combining the texture, shading and transparency.

Most of this demo was programmed using *C++* except for the rasterization core coded in assembler.

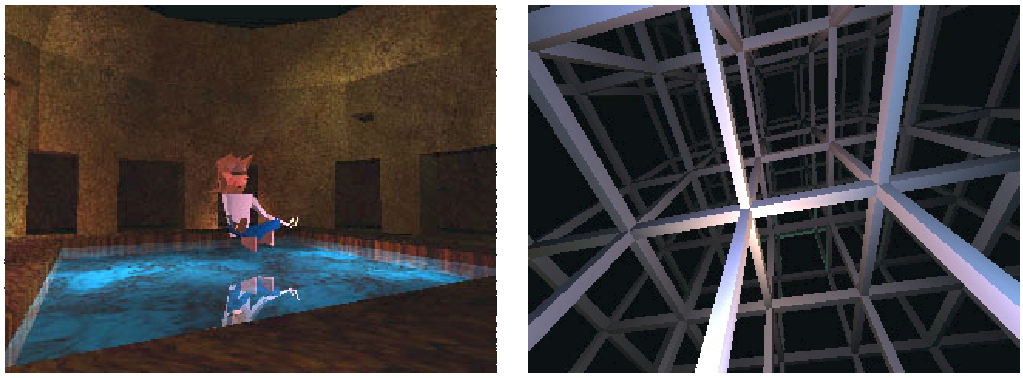


Figure 6: The left screenshot is an example of scene graph rendering, while the right one shows a flight through an endless 3D grid, both featured in the *Dream* demo.

4.2 EXPIRATION demo

Expiration [17] was created one year after the *Dream* demo, and released again at the *Fiasko* demoparty. The platform of our choice changed to *Microsoft Windows* where we also benefited from hardware acceleration through *OpenGL*. These choices led to a creation of a very stable production as opposed to *Dream*, with no hardware compatibility problems that we know of. *Expiration* placed first out of the other 15 contributions in the demo competition.

In the following sections we will present the behinds of two interesting parts of our demo: *train track* and *bouncing light sphere*.

Train track

At this point we will cover the method for creation of train track models that we've utilized for building up the rollercoaster part of *Expiration*.

The primary goal we set up was to design a method that would allow us to *create* and *manipulate* the track shape with ease and at the same time provide simple ways of *placing* vehicles – in our case a locomotive followed by the train's waggons – onto the track.

Two basic ways suggesting themselves were to let the work be done by a human 3D artist in a modelling tool or to take a more challenging approach of describing

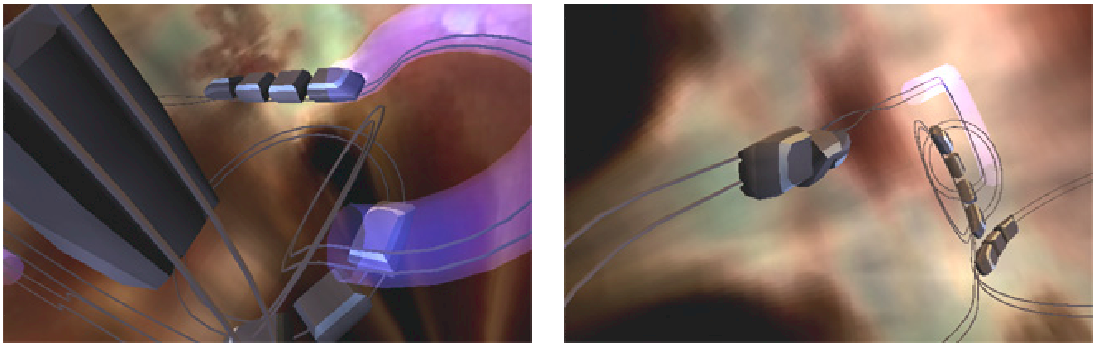


Figure 7: Screenshots from the rollercoaster part of *Expiration*.

the track's shape in a mathematical way, possibly as a parametric spline and building the track's 3D mesh along the spline.

Although the former alternative of leaving the work up to an artist has the advantage of better control over the appearance of the track's mesh and the possibility of giving the model a kind of an artistic touch; it suffers from two rather unsatisfying properties. Modifications to the track's shape require the artist to make changes to the work already done, which may take time and a considerable effort with the number of modifications increasing. The second fact concerns the problem of placement of the vehicles onto the track. Clearly there is no way that would produce naturally looking results, without additional work for the 3D artist to maintain the animation paths that would capture the track's shape and orientation. These disadvantages led us to taking a more in-depth look at the qualities of the latter alternative mentioned, hence describing the track in a *parametric* way.

Parametric track representation

Here we'll present some basic observations and formulate requirements that the parametric representation of a track must meet.

As a first observation, it's clear that what our representation must capture is not only the track's shape, but also its *orientation varying* along, to be able to properly build the final mesh and to allow simple vehicle placement.

Concerning only the shape, there exist plenty of successful methods applicable for describing it, such as straight lines, Bézier curves, B-splines or arcs. The track in *Expiration* was built using segments based on variations or little modifications of *arcs* and *straight lines*. While being not so flexible as Bézier curves or B-splines or even NURBS, they still have a number of properties favourable to our problem. In the real world the railway's or tram's tracks are not arbitrary, they are mostly straight and their turns resemble arcs. Another rather practical fact is that straight lines and arcs can be evaluated at even distances by simply uniformly substituting for the parameter – as opposed to Bezier curves or B-splines, where evaluation at even distances would require approximation – which is useful for quick vehicle placement and mesh generation.

Track segment

Let us describe the track more formally. As already said, our formalization must parametrize shape and orientation. Since we are going to build the track from segments, some simple means of joining the segments together will be required. With the neccessity of segment joining in mind, to describe the track's location and orientation at each parameter value, we have chosen homogeneous 4×4 matrices consisting of only rotation and translation.

Let's define the track segment's parametric representation as:

$$s : [0, 1] \to \mathcal{R}_{4\times4}$$

with $[0, 1]$ denoting the interval of real numbers between 0 and 1 inclusive and $\mathcal{R}_{4\times4}$ denoting the group of 4×4 matrices representing *rigid body transforms* of the Euclidean 3-space in homogeneous form, effectively being just a set of all matrices representing *rotation* and *translation* of the 3-space. Thus:

$$s : t \mapsto \begin{pmatrix} d_x & u_x & r_x & p_x \\ d_y & u_y & r_y & p_y \\ d_z & u_z & r_z & p_z \\ 0 & 0 & 0 & 1 \end{pmatrix},$$

where $d = (d_x, d_y, d_z)^T$ is a directional vector pointing in *front* of the track's orientation, $u = (u_x, u_y, u_z)^T$ represents the *up* vector, and $r = (r_x, r_y, r_z)^T$ is the vector to the *right* of the orientation; clearly the (d, u, r) triple must be orthonormal for the matrix to represent rigid body transform. Finally, $p = (p_x, p_y, p_z)^T$ is the position of the track.

Function s captures the track's segment's position and orientation at every parameter t in the interval $[0, 1]$, with $s(0)$ representing the beginning of the track segment and $s(1)$ being the end of the track segment. Thus by projecting off the rotational part of the matrix and concerning just the translation we could get a parametric vector function describing the segment's shape.

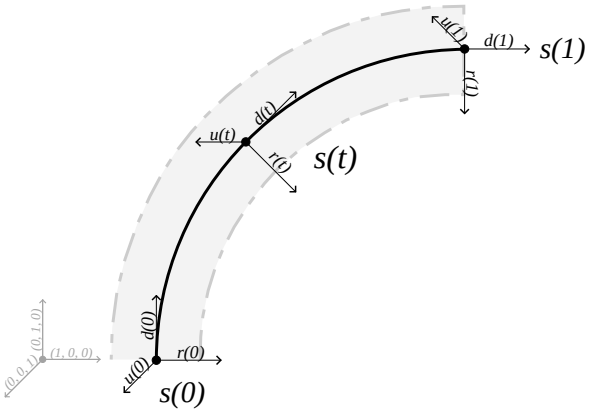


Figure 8: A segment with the *direction*, *up* and *right* vector functions.

To avoid more complicated joining of segments, we will assume that for every segment s it holds that:

$$s(0) = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Obviously, function s could be used as a guide for building up the 3D mesh around the track segment, but we can also utilize it for the vehicle placement, since with function s we know everything about the position and orientation of the track segment at any desired parameter value t.

As an example, this is a turn right by 90° degrees:

$$\begin{pmatrix} \sin(\psi(t)) & 0 & -\cos(\psi(t)) & R(1+\cos(\psi(t))) \\ -\cos(\psi(t)) & 0 & \sin(\psi(t)) & R\sin(\psi(t)) \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

with $\psi(t) = \pi - \frac{\pi}{2}t$ and R being the radius of the turn.

Entire track

The entire track T consists of a sequence of track segments $s_1, s_2, \ldots, s_n$ with their respective geometric lengths $l_1, l_2, \ldots, l_n$. Now we can define the entire track as a sequence of $(segment, length)$ pairs:

$$T = \{(s_i, l_i)\}_{i=1}^{n},$$

where every s_i is a function of the form:

$$s_i(t) = \begin{pmatrix} d_i(t) & u_i(t) & r_i(t) & p_i(t) \end{pmatrix}.$$

Now we'll define a *knot sequence* $t_0, t_1, \ldots, t_n$ we're about to use in the definition of the *entire track evaluation function* afterwards. Every knot t_i will represent a parameter value for the evaluation function at the joint point of two successive track segments.

$$t_k = \begin{cases} 0, & \text{if } k = 0 \\ \frac{1}{length(T)} \sum_{i=1}^{k} l_i, & \text{if } 1 \leq k \leq n \end{cases}$$

With $length(T)$ we denote the geometric length of the entire track, more precisely:

$$length(T) = \sum_{i=1}^{n} l_i \,.$$

Finally, using the knot sequence given we will bring in the function S that represents the joining of successive track segments, being a function evaluating the

entire track's position and orientation along, with $S(0)$ describing the beginning of the track, and $S(1)$ being the end of its last segment.

$$S : [0, 1] \to \mathcal{R}_{4\times 4}$$

$$S(t) = \begin{cases} S(t_{i-1})s_i(\frac{t-t_{i-1}}{t_i-t_{i-1}}), & \exists 1 \leq i \leq n : t \in (t_{i-1}, t_i] \\ s_1(0), & \text{otherwise } (t = t_0) \end{cases}$$

Camera views

There were two sorts of camera views utilized within the rollercoaster part of the *Expiration* demo: cameras focused on and moving next to one of the trains, and free, unfocused cameras giving views of a major part of the entire track.

Each of the two were accomplished by employing *spherical linear interpolation* of *quaternions* [18], more precisely:

$$slerp(t,\, q_0,\, q_1) = \frac{q_0 \sin((1-t)\theta) + q_1 \sin(t\theta)}{\sin\theta}$$

with θ being the angle between the unit quaternions q_0 and q_1.

The unfocused camera views were realized by simply interpolating two fixed camera positions and orientations during a predefined time period, while the other sort of camera views focused on a moving train were slightly more complicated to do, and we are going to describe the method used shortly.

Let's say that the train we would like to focus our camera on is situated somewhere between $S(t_e)$ and $S(t_b)$, for some $t_e < t_b$. We could simply choose a parameter value u such that $t_e \leq u \leq t_b$, and use the $S(u)$ value as a coordinate space basis useful for placing our camera at $S(u) * (C_x,\, C_y,\, C_z,\, 1)^T$ for some position C and targeted at the point written in the last column vector of $S(u)$ matrix, which effectively is track's position at parameter value u. This way the camera would fly next to the train, but with its motion precisely copying the track's shape, which looks much too artificial to be convincing.

We will avoid the artificial look of previosly described approach using quaternion interpolation. Analogously to the function S we could define another function:

$$Q : [0, 1] \to \hat{H}$$

that would capture the track's orientation all along using unit quaternions instead of matrices.

By spherical linear interpolation of quaternions $Q(t_e)$ and $Q(t_b)$, for example $q_{mid} = slerp(\frac{1}{2},\, Q(t_e),\, Q(t_b))$ we would get an orientation that resembles the track's orientation as the train moves. It wouldn't be an exact copy but a harsh approximation depending on how we choose the t_e and t_b parameter values. That would give a more natural look to the camera views generated analogously to the technique described before, after conversion of q_{mid} to matrix M.

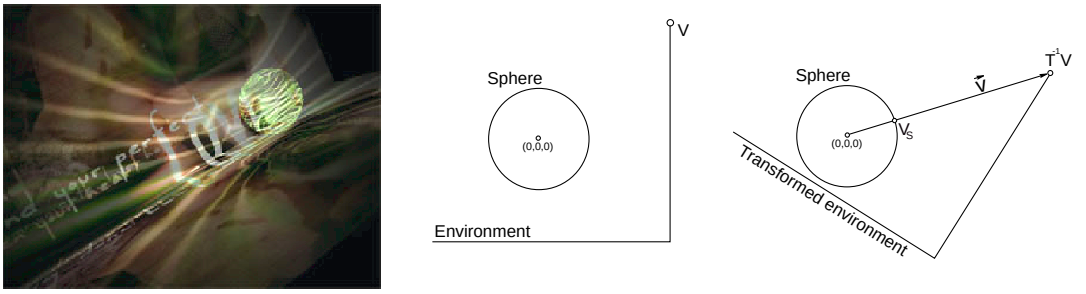
We could further place the camera not at a fixed position relatively to the M basis, but let this placement be a function of time, placing the camera at $M * C(t)$, and same with the camera's target.

Bouncing Light Sphere

This effect appearing at the end of *Expiration* was achieved by center projection of the surface texture of a projecting object on its *environment* utilizing texture mapping [19]. In the method presented, the projecting object is restricted to possess such a shape where it is possible to calculate an intersection with a ray easily. On the other hand, the *environment* around the projecting object is allowed to be rather complicated. For better results, the environment mesh should by sufficiently tesselated which would help eliminating the effect of non-linearity of center projection in the plane of projection. That's why the two planes in the demo are actually grids. During the rendering, both the object and its *environment* are represented as a triangle mesh.

For the projecting object we choosed an *unit sphere* initially located at the origin $o = (0,\ 0,\ 0)$ of the 3D space. We define a function $\tau : R^3 \to R^2$ to express the texture coordinates across the sphere surface, to be more precise $\tau : P \mapsto (u,\ v)$ for any point P on the surface.

The effect of projecting the *sphere's* surface on the *environment* was realized during an extra rendering pass, where texture of the sphere was mapped onto the environment and texture coordinates for each of *environment*'s vertices were calculated according to position and rotation of the *sphere* in the following way.



(a) (b)

Figure 9: (a) Screenshot from demo. (b) Obtaining a point on the sphere surface.

Let the *sphere*'s animated position and rotation be expressed by some transformation T. The whole scene is transformed using T^{-1}, to relocate the sphere back to its initial position and rotation at the origin of coordinate space for making the further calculations easier. So, for each vertex $V = (x_V,\ y_V,\ z_V,\ 1)^T$ of the *environment* the following vector is found:

$$\vec{u} = T^{-1} \begin{bmatrix} x_V \\ y_V \\ z_V \\ 1 \end{bmatrix} - \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}$$

Thanks to the *sphere*'s position and radius chosen, the intersection point of a ray $R(t) = T^{-1}V + t * \vec{u}$ with the *sphere*'s surface can be found rapidly:

$$V_S = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} + \frac{\vec{u}}{\| \vec{u} \|}$$

Finally, this chosen vertex V, texture coordinates are then $\tau(V_S)$ and these same coordinates are used during an extra pass of *environment* rendering with the sphere's texture.

4.3 Symbolic Expression 4KB intro

Symbolic Expression [20] was presented at *Fiasko* 2001 in Czech Republic. Four kilobytes is not much, and there are usually two ways to go – either extensive size optimization or an elegant idea. This intro is an example of the second approach. The intention was to create a few colorful 2D texture effects using OpenGL, taking advantage of hardware acceleration. While analyzing possible solutions, one could note that in fact any 2D effect can be thought of as a single function assigning a color triple (r, g, b) to every screen coordinate (x, y) at time τ:

$$e : R^3 \to R^3 \qquad e(x, y, \tau) = (r, g, b)$$

In many cases, this function can be decomposed into $e = t \circ m$, where the m function is a mapping from screen coordinates (and time) into texture coordinates and the t function specifies the texture:

$$m : R^3 \to R^2 \qquad m(x, y, \tau) = (u, v)$$

$$t : R^2 \to R^3 \qquad t(u, v) = (r, g, b)$$

In theory, the whole demo could be written as a single effect function. This function could be defined as an algebraic expression involving the variables x, y and τ – hence the name of the intro.

These ideas are very simple to implement. The texture and mapping functions for different effects can be written directly in the C language. The textures are evaluated and sent to OpenGL at the initialization of the intro. The screen is subdivided into small squares and the mapping function is evaluated at every corner of the grid in a given time moment τ. Each square is filled with a textured OpenGL quad primitive. The alpha-blending feature of OpenGL allows for several "summed" effects on the screen – the intro uses three to six layers.

The only question that remains unanswered is how to define the mapping and texture functions. As for the mapping, it can be defined as a composition of several functions such as affine transforms, "caleidoscopic" mapping

$$cal(x, y) = (\min(|x|, |y|), \max(|x|, |y|)),$$

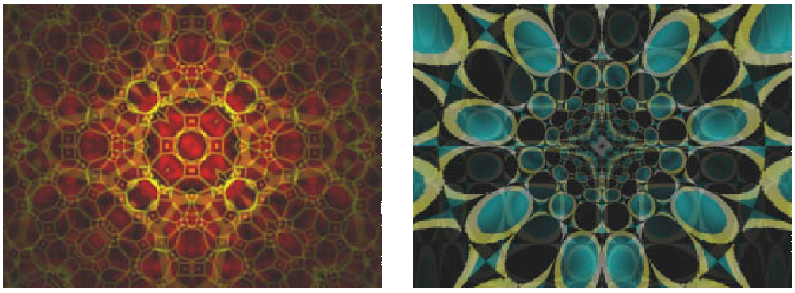


Figure 10: Screenshots from the intro.

sine distortion, e. g.

$$sd(x, y, \tau) = (ax + b\sin(y + \tau), ay + b\cos(x + \tau)),$$

or anything that comes to mind and looks good. To define the texture functions, the intro uses functional representation (F-rep) [21]. A function $f : R^2 \to R$ defines the 2D set $\{(x, y)|f(x, y) \geq 0\}$. By combining linear, quadratic or other curves with the functions min and max (as intersection and union), different 2D shapes can be described by very few lines of code. With the help of a noise function and different colors, we get the actual RGB texture functions.

5 Conclusion

This paper introduced the *demoscene* community phenomenon with its history and present trends with emphasis on bindings to computer graphics and computer programming. The authors further presented some of their own work in this area, describing several methods they used during the creation of the discussed demos.

In the end we would like to give our thanks to all the reviewers for their valuable advices, helping to improve the quality of our paper.

References

[1] MÖLLER, T. – HAINES, E. 1999. *Real-time Rendering.* A K Peters, Ltd., 1999, second edition. ISBN 1-56881-101-2. `http://www.realtimerendering.com`.

[2] *The scene.org demoarchive.* `http://www.scene.org`.

[3] *256 byte intros.* `http://www.256b.com`.

[4] *Assembly.* `http://www.assembly.org`.

[5] *The Party.* `http://www.theparty.dk`.

[6] *Mekka-Symposium.* `http://ms.demo.org`.

[7] *Commodore C-64 computer.* http://www.c64.org.

[8] AMIGA *computer.* http://www.amiga.com.

[9] *Intel x86 processor series.*
http://developer.intel.com/design/processor.

[10] BLOOMENTHAL, J. Polygonization of implicit surfaces. *Computer Aided Geometric Design*, 5(4) pp. 341–355, Nov 1988.

[11] WALD, I. – SLUSALLEK, P. State-of-the-art in interactive ray-tracing. In *State of the Art Reports, EUROGRAPHICS 2001*, pp. 21–42, Sep 2001.

[12] GOOCH, B. – GOOCH, A. A. 2001. *Non-Photorealistic Rendering.* A K Peters, Ltd., 2001. ISBN 1-56881-133-0.

[13] STROUSTRUP, B. 2000. *The C++ Programming Language Special Edition.* Addison-Wesley, 2000. ISBN 0-201-88954-4.

[14] OpenGL Architecture Review Board. 1999. *OpenGL Reference Manual 3rd edition.* Addison-Wesley, 1999. ISBN 0-201-65765-1.

[15] *Microsoft DirectX.* http://www.microsoft.com/directx.

[16] PAULOVIČ, O. – BURGER, B. – PLACHÝ, P. – RUTTKAY, L. 1999. *Dream.*
ftp://ftp.no.scene.org/scene.org/parties/1999/fiasko99/demo/dream.zip.

[17] BURGER, B. – PAULOVIČ, O. – PLACHÝ, P. – RUTTKAY, L. 2000. *Expiration.*
ftp://ftp.no.scene.org/scene.org/parties/2000/fiasko00/demo/expirati.zi%p.

[18] DAM, E. B. – KOCH, M. – LILLHOLM, M. Quaternions, interpolation and animation. Technical Report DIKU-TR-98/5, Department of Computer Graphics, University of Copenhagen, July 1998.

[19] HAEBERLI, P. – SEGAL, M. Texture mapping as a fundamental drawing primitive. In *Fourth Eurographics Workshop on Rendering*, pp. 259–266, Jun 1993.

[20] HAŠAN, M. 2001. *Symbolic expression.*
ftp://ftp.no.scene.org/scene.org/parties/2001/fiasko01/in4k/symbl.zip.

[21] PASKO, A. – ADZHIEV, V. – SOURIN, A. – SAVCHENKO, V. Function representation in geometric modeling: concepts, implementation and applications. *The Visual Computer*, 11(8), 1995.