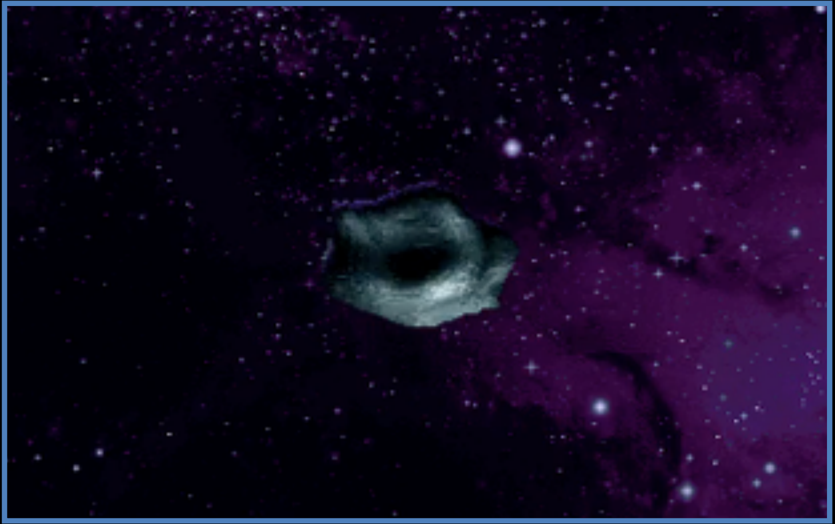
EMBERS BY TDA 2012

What? Why? And more importantly, how?...

WHAT IS DEMOSCENE?

The demoscene is a computer art subculture that specializes in producing demos, which are non-interactive audio-visual presentations that run in real-time on a computer. The main goal of a demo is to show off programming, artistic, and musical skills.

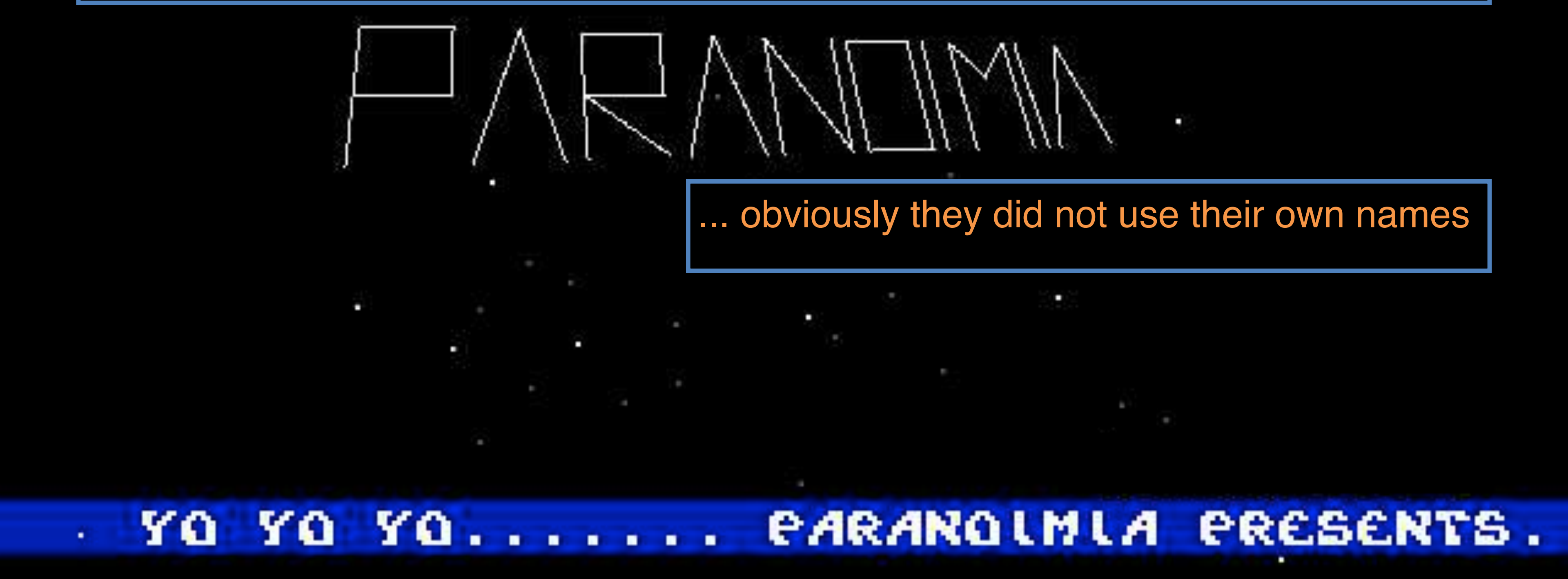
-- wikipedia


Dope by Complex, 1996

fr-041: debris by Farbrausch, 2007

ORIGINS/HISTORY

Back in time when computer software was still delivered by floppies or by dial-by-modem BBSes, an underground movement was born which specialized pirating commercial software and games. Later those individuals organized themselves to groups, and they started to add splashscreen (a.k.a. cracktros) to SW they pirated to advertise their BBS and take credit for the work they did...

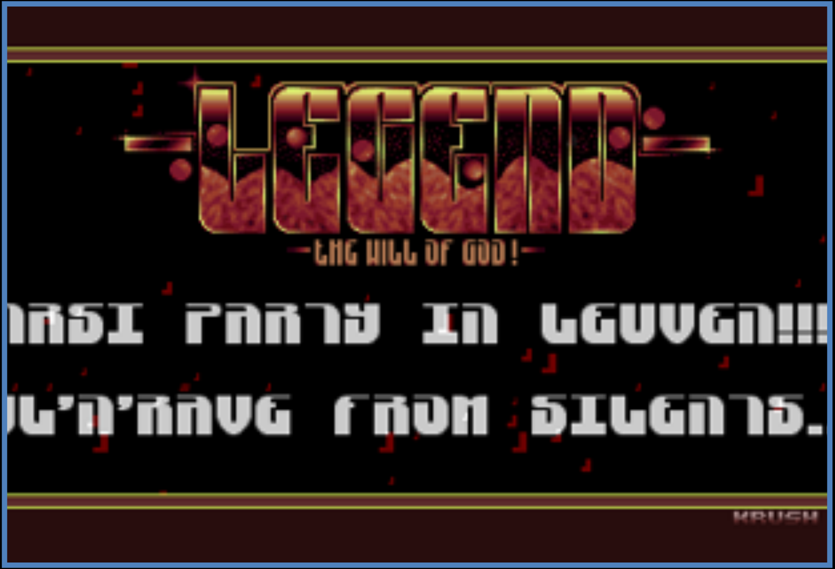


... obviously they did not use their own names

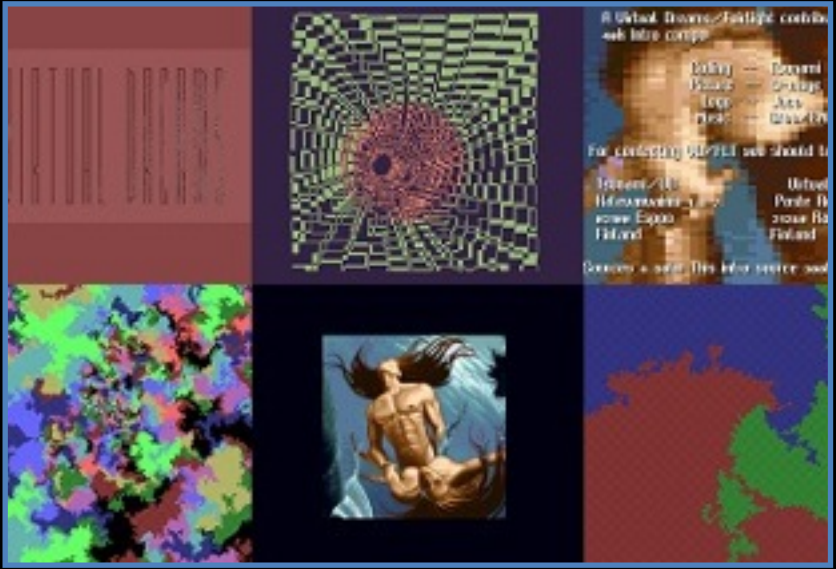
Paranoimia cracktro
c.a. 1989

ORIGINS/HISTORY

Some groups started competing on making the best cracktros which later grew to standalone intros and demos and this attracted more individuals which were interested about the computer graphics and audio but were shunned away from the criminal element. The crack groups and demo groups separated...



New cracktro by Legend and Origin, 1992



40k intro by Virtual Dreams & Fairlight, 1993

COMPETITIONS

The early gatherings i.e. “party” mainly composed of distributing pirated software i.e. “warez”.

Some of these later transformed into demoparties where intros/demos competed on different categories on all imaginable hardware.

In early days best productions presented what was thought to be “impossible” at the time, however now in the PC-era size-limited competitions have gained popularity since hardware is flexible...

We participated to 1K-intro compo Assembly 2012 with our product...

1 K COMPO RULES

Rules are straightforward:

- The competition entry has to be executable file, maximum size 1024 bytes
- It has to run on modern operating system (Win, Mac, Linux or similar)
- Fresh install of operating system, updates applied. No internet connectivity.
- (All other files from the package deleted but this executable)
- Has to run on certain resolutions (f.e. 1280x720, 800x600)
- It must be able to run from read-only media, may use system temporary directory
- It must hide mouse cursor
- It must quit with ESC-key
- Maximum running time: 2 minutes

We chose Mac OS X/OpenGL and this is how we used our 1024 bytes...

EMBERS / TDA / 2012

HOW?

BASICS

Making executables that are less than a kilobyte and still useful might feel like a impossible task since...

```
$ cat > test.c
#include<stdio.h>
int main(){printf("Hello world\n");return 0;}
$ gcc test.c ; strip a.out
$ wc -c a.out
    8552 a.out
```

However, this is not the limit because we can handcraft the headers.

It is well known that smallest Linux ELF-binary can be as small as 45 bytes, and Mac OS X mach-o binary as small as 165 bytes. But...

These binaries are as irrelevant as that 8552-byte hello world.

The real question is: How low you can go with a dynamic linking executable i.e. executable that links to OpenGL, DLSSynth and other system libs?

DYNAMIC LINKING

References to library functions consume a lot of space. At bare minimum you need strings + symtab + DYLIB command for each lib.

Fortunately there are techniques to reduce this waste. We can put the library references to executable headers but leave symtab and strings out.

This will make the system treat the executable like static binary, with one crucial difference: It loads the libraries asked and dynamic loader is present.

However, can we solve the library references more efficiently than the systems dynamic loader?

Yes we can!

DYNAMIC LINKING

Steps to do:

1. Resolve internal relocations in the code by using pic-code and/or linking to the static address
2. Map all the external references through a trampoline. (aka. PLT)
3. Find a system library the depends to the all external libraries needed.
   - In Mac OS X good choice is either Carbon Or Cocoa.

Now binary does not have any relocations and all the external references are in a linear array in PLT which we need to populate.

We could easily dlsym() all the funcs to PLT (if we would know where dlsym is located), however there is even better way...

void *__dyld_get_image_header(int index) returns address of loaded library by using indexing through all the loaded libraries.

And dyld is located at 0x8fe00000 + ASLR fudge. So lets go to the libraries directly and skip dlsym completely.

And as an added bonus: when loading the references we do not need to match strings only hashes we have calculated from the strings. This reduces size of the external reference to 3 bytes total.

LOAD COMMANDS

```
Load command 0
        cmd LC_UNIXTHREAD
    cmdsize 80
     flavor i386_THREAD_STATE
      count i386_THREAD_STATE_COUNT
          eax 0x00000000 ebx    0x00000000 ecx 0x00000000 edx 0x00000000
          edi 0x00000000 esi    0x00000000 ebp 0x00000000 esp 0x01000000
          ss  0x00000000 eflags 0x00000000 eip 0x000000ee cs  0x00000000
          ds  0x00000000 es     0x00000000 fs  0x00000000 gs  0x00000000
Load command 1
      cmd LC_SEGMENT
  cmdsize 56
  segname
   vmaddr 0x00000000
   vmsize 0x01000000
  fileoff 0
 filesize 2145
  maxprot ---
 initprot rwx
   nsects 0
    flags (none)
Load command 2
          cmd LC_LOAD_DYLINKER
      cmdsize 28
         name /usr/lib/dyld (offset 12)
Load command 3
          cmd LC_LOAD_DYLIB
      cmdsize 48
         name Cocoa.framework/Cocoa (offset 24)
   time stamp 0 Thu Jan  1 02:00:00 1970
      current version 0.0.0
compatibility version 0.0.0
```

COMPRESSION

For size limited intros nothing is so important as good compression.

4K intros (and larger) usually utilize state of the art compressors with hand-crafted assembly-optimized decompressors. This is not feasible for 1K because there would be too much overhead.

- Windows 1Ks either get DirectX zlib calls, or use small custom compressor
- Unix like system utilize shell script decompressors aka. shell droppers.

Markku "Marq" Reunanen presented his shell dropper 2006, which was the starting point of dropper development for unix-like systems. (56 bytes + 10 byte gzip header)

```
a=/tmp/I;tail -n+2 $0|zcat>$a;chmod +x $a;$a;rm $a;exit
GZIPHEADERS
```

We have our version of shell dropper with Embers. It takes advantage of the fact that there is unnecessary data in gzip header. (34 bytes + 10 bytes gzip header)

```
cp $0 /tmp/z;(sed 1d $0|zcat
GZIP)>$_;$_
)
0
```

This is a technique only we are using so far and winning ~20-bytes by doing so

COMPRESSIBLE CODE

After happily implementing a compressor (and decompressor) a naïve approach would be to happily program a smallest possible program and let compressor to make it a wee bit smaller...

This is totally wrong approach

All compressors independent of actual algorithm used, use the input data to match patterns, and utilize them to code the output.

So if you have a small binary which is handcrafted algorithmically to be as tight as possible, there is no data nor patterns and your result is most likely expanding.

Repetition is the key

Unrolling loops, modifying constants, applying extra parameters, changing behavior slighly and all other dirty tricks when it makes the compressed code smaller without paying attention to uncompressed size is the right approach.

Uncompressed size of Embers is 2145 bytes (45,6% compression ratio)...

EXAMPLES

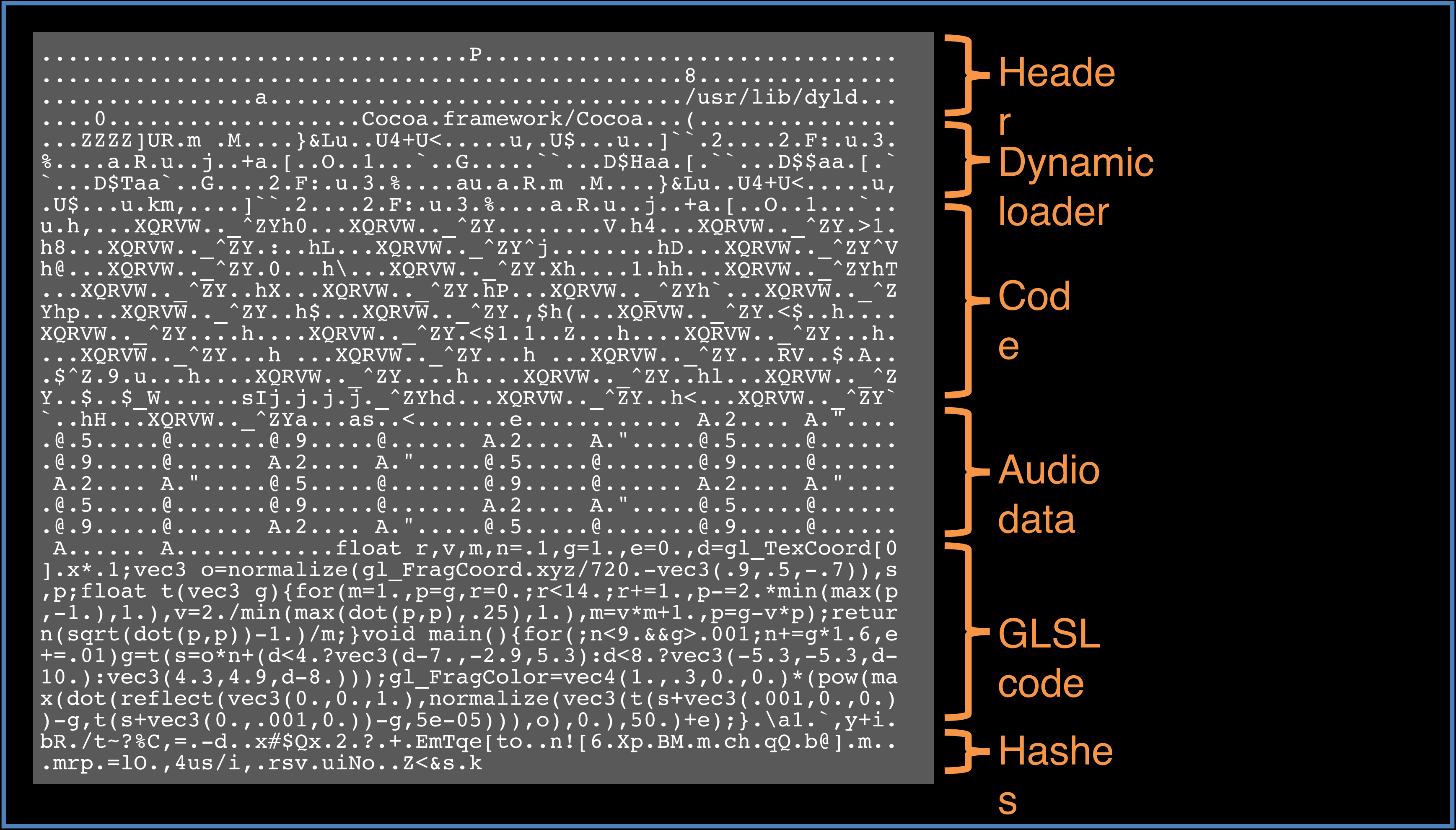
Bad stuff:

```
int f(int x) {return x*x+7;}


for (int x=0;x<3;x++) f(x);


glRectf(-1.0,-1.0,1.0,1.0);



glVertex2i( 1, 1);
glVertex2i( 1,-1);
glVertex2i(-1,-1);
glVertex2i(-1, 1);


glTexCoord2f(x,y);
glVertex3f(x,y,z);
```

Better stuff:

```
#define f(x) ({int _x=x;_x*_x+7;})


int x=0;f(x++);f(x++);f(x++);


glRectf(-2.0,-2.0,2.0,2.0);


int x=1,y=1,tmp;
glVertex2i(x,y);tmp=x;x=y;y=-tmp;
glVertex2i(x,y);tmp=x;x=y;y=-tmp;
glVertex2i(x,y);tmp=x;x=y;y=-tmp;
glVertex2i(x,y);tmp=x;x=y;y=-tmp;


glTexCoord3f(x,y,z);
glVertex3f(x,y,z);
```

UNCOMPRESSED CODE DUMP



```
................................P...............................
................................................8...............
................a.............................../usr/lib/dyld...
....0...................Cocoa.framework/Cocoa...(...............
...ZZZZ]UR.m .M....}&Lu..U4+U<.....u,.U$...u..]``.2....2.F:.u.3.
%....a.R.u..j..+a.[..O..1...`..G.....``...D$Haa.[.``...D$$aa.[.`
`...D$Taa`..G....2.F:.u.3.%....au.a.R.m .M....}&Lu..U4+U<.....u,
.U$...u.km,....]``.2....2.F:.u.3.%....a.R.u..j..+a.[..O..1...`..
u.h,...XQRVW.._^ZYh0...XQRVW.._^ZY........V.h4...XQRVW.._^ZY.>1.
h8...XQRVW.._^ZY.:..hL...XQRVW.._^ZY^j........hD...XQRVW.._^ZY^V
h@...XQRVW.._^ZY.0...h\...XQRVW.._^ZY.Xh....1.hh...XQRVW.._^ZYhT
...XQRVW.._^ZY..hX...XQRVW.._^ZY.hP...XQRVW.._^ZYh`...XQRVW.._^Z
Yhp...XQRVW.._^ZY..h$...XQRVW.._^ZY.,$h(...XQRVW.._^ZY.<$..h....
XQRVW.._^ZY....h....XQRVW.._^ZY.<$1.1..Z...h....XQRVW.._^ZY...h.
...XQRVW.._^ZY...h ...XQRVW.._^ZY...h ...XQRVW.._^ZY...RV..$.A..
.$^Z.9.u...h....XQRVW.._^ZY....h....XQRVW.._^ZY..hl...XQRVW.._^Z
Y..$..$_W......sIj.j.j.j._^ZYhd...XQRVW.._^ZY..h<...XQRVW.._^ZY`
`..hH...XQRVW.._^ZYa...as..<.......e............ A.2.... A."....
.@.5.....@.......@.9.....@...... A.2.... A.".....@.5.....@......
.@.9.....@...... A.2.... A.".....@.5.....@.......@.9.....@......
 A.2.... A.".....@.5.....@.......@.9.....@...... A.2.... A."....
.@.5.....@.......@.9.....@...... A.2.... A.".....@.5.....@......
.@.9.....@...... A.2.... A.".....@.5.....@.......@.9.....@......
 A...... A............float r,v,m,n=.1,g=1.,e=0.,d=gl_TexCoord[0
].x*.1;vec3 o=normalize(gl_FragCoord.xyz/720.-vec3(.9,.5,-.7)),s
,p;float t(vec3 g){for(m=1.,p=g,r=0.;r<14.;r+=1.,p-=2.*min(max(p
,-1.),1.),v=2./min(max(dot(p,p),.25),1.),m=v*m+1.,p=g-v*p);retur
n(sqrt(dot(p,p))-1.)/m;}void main(){for(;n<9.&&g>.001;n+=g*1.6,e
+=.01)g=t(s=o*n+(d<4.?vec3(d-7.,-2.9,5.3):d<8.?vec3(-5.3,-5.3,d-
10.):vec3(4.3,4.9,d-8.)));gl_FragColor=vec4(1.,.3,0.,0.)*(pow(ma
x(dot(reflect(vec3(0.,0.,1.),normalize(vec3(t(s+vec3(.001,0.,0.)
)-g,t(s+vec3(0.,.001,0.))-g,5e-05))),o),0.),50.)+e);}.\a1.`,y+i.
bR./t~?%C,=.-d..x#$Qx.2.?.+.EmTqe[to..n![6.Xp.BM.m.ch.qQ.b@].m..
.mrp.=lO.,4us/i,.rsv.uiNo..Z<&s.k
```

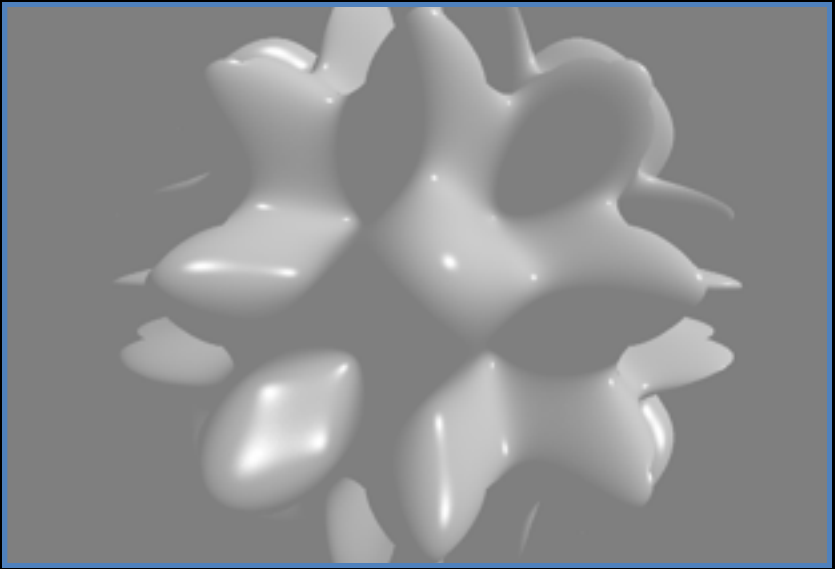
GRAPHICS

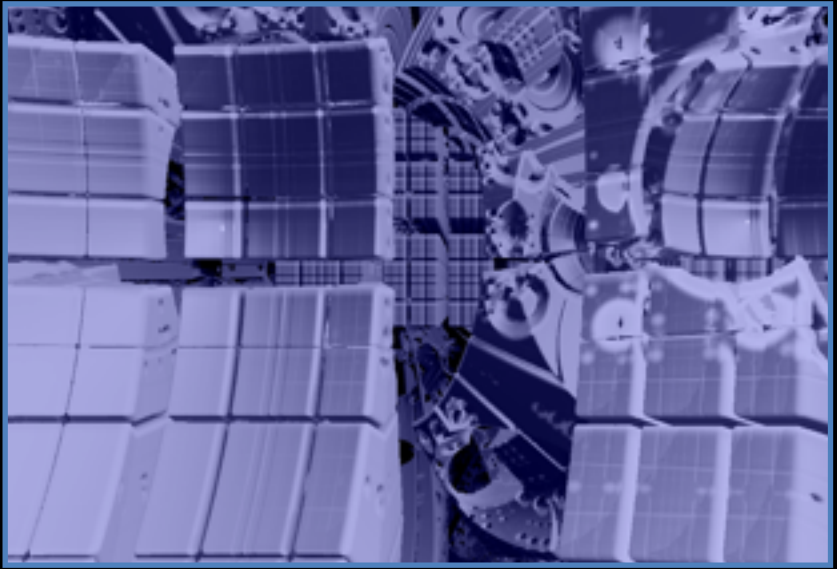
SO, WHAT COULD WE DO?

Next step was lots prototyping and decide what we can do. We divided the budget into three parts.

Effect: 600 bytes, Audio 200 bytes, linker and overhead 200 bytes

For so little code for effect we tested and determined that it has to be shader-only, preferably raytracing something nice...


Testing ray casting of a simple object


Testing distance function of a fractal

GLSL INTRO BASICS

To get most of the size-limited code, usually the complete effect is done with shaders. The GLSL code is stored as a text which compresses well and there is not usually need to optimize the code for speed...

What is left for the CPU-side code is to provide a timestamp “ticker” and render a screen sized quad where our shader program is bound.

When raymarching/raytracing vertex shader can be omitted and all the functionality written into fragment shader directly.

OpenGL 2.x is better suited to the job than OpenGL 3.x+
(Immediate mode helps setup and initialization code is smaller)

UNIFORMS

The fragment shader needs uniform: time i.e. “ticker”

There are three ways to get it to fragment shader (no vertex shader)

1. glUniform, the official way to do it
   - Needs many library calls to setup. And definition in the GLSL-code which means code will get larger.
2. glColor
   - Practical way of getting uniform (and has integer input as well with glColor3s()). however, input is clamped.
3. glTexcoord
   - Perfect easy way to get one uniform to shader, assuming texture coordinates are not used...

FRAGMENT COORDINATES

The second parameter that is needed is the fragment (x,y) coordinates. There are two ways to get these.

1. Use screen coordinates “gl_FragCoord.xy”
   - Pros: Readily available.
   - Cons: Needs to be scaled to the size of the drawable i.e. screen
2. Use texture coordinates “gl_TexCoord[0].xy”
   - Pros: When set together with quad-coordinates, they are independent of the screen size...
   - Cons: ...but still dependent of the screen aspect ratio

CODE FOR OPENGL

```
CGCaptureAllDisplays();
CGDisplayHideCursor(0);

CGLContextObj ctx;CGLPixelFormatObj pf;Glint dummy;
CGLPixelFormatAttribute attr[] = {kCGLPFADoubleBuffer,0};
CGLChoosePixelFormat(attr,&pf,&dummy);
CGLCreateContext(pf,0,&ctx);
CGLSetCurrentContext(ctx);
CGLSetFullScreenOnDisplay(ctx,0);

GLuint shader=glCreateShader(GL_FRAGMENT_SHADER);
glShaderSource(shader,1,&shader_src,0);
glCompileShader(shader);
GLuint program=glCreateProgram();
glAttachShader(program,shader);
glLinkProgram(program);
glUseProgram(program);

char keys[16];
do {
     glTexCoord1dv(&ticker);
     glRecti(-1,-1,1,1);
     CGLFlushDrawable(ctx);
     GetKeys((KeyMap)keys);
} while (!(keys[6]&20));
```

Grab displays, hide cursor

Any format, GL-context

Create our shader

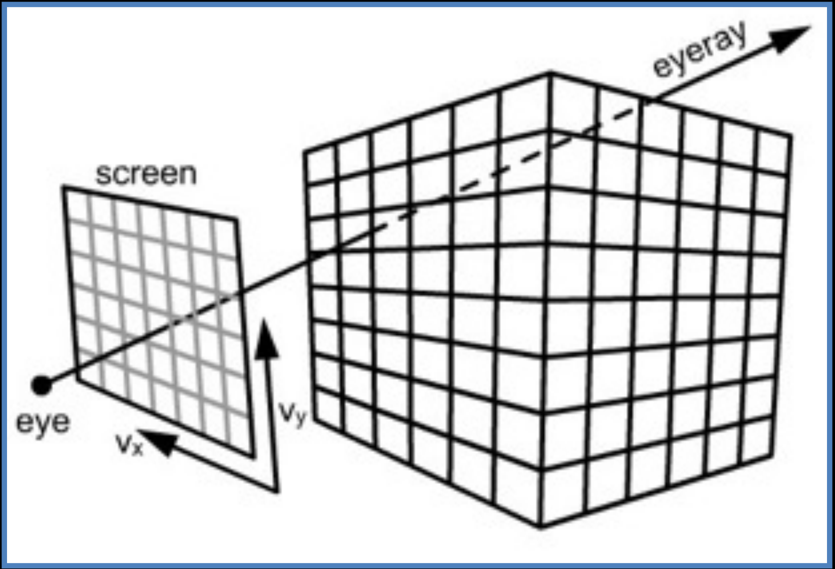
No clear, just draw quad

Detect ESC-key

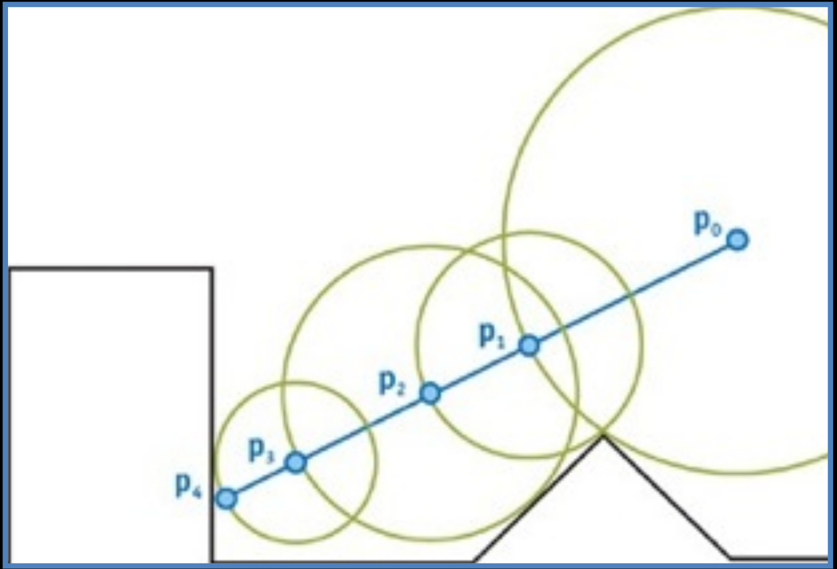
RAY TRACING/CASTING

In very simplified form of ray tracing light path vector is generated using eye coordinate and screen plane and stepping it until surface is found or iteration limit is exceeded. This is either very slow (small step) or very inaccurate (large step)...

Better way is to use a distance function (assuming it exists for the object being raytraced). As the name implies it returns shortest distance to the object in question. This way size of the step is always optimal.



Ray casting



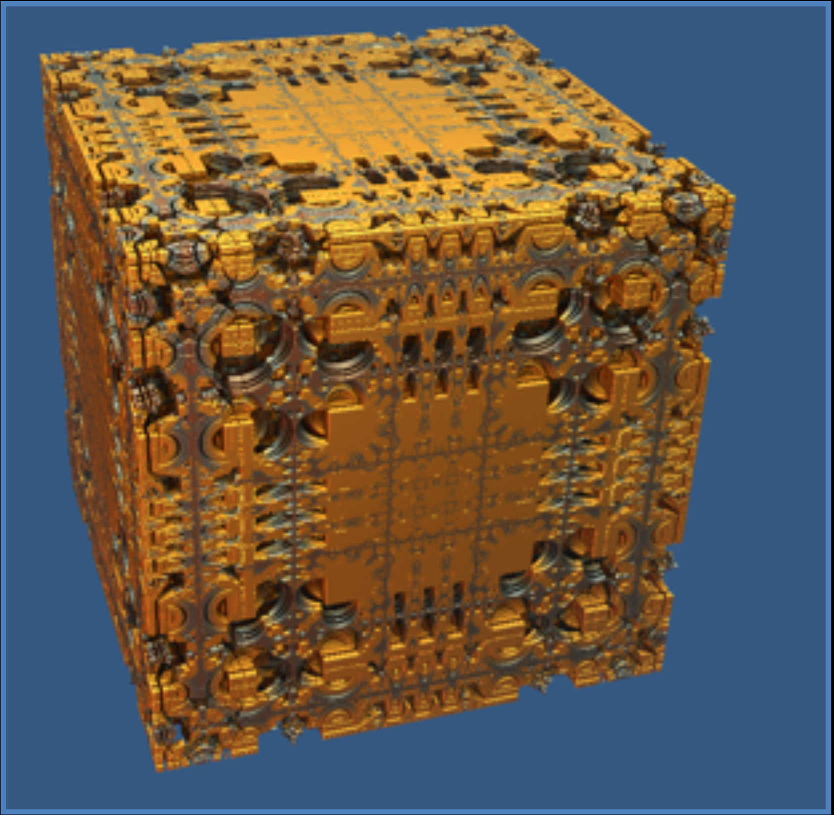
Distance function

MANDELBOX

Depending on the scale-value of a mandelbox fractal it can look like organic, industrial or anything between. It has a simple definition for both fractal and distance function:

```
function iterate(z):
  for each component in z:
    if component > 1:
      component := 2 - component
    else if component < -1:
      component := -2 – component
  if magnitude of z < 0.5:
    z := z * 4
  else if magnitude of z < 1:
    z := z / (magnitude of z)^2
  z := scale * z + c
```

The only real downside is the iterative nature of the fractal which makes it time consuming to raytrace



Scale 2 Mandelbox (Wikipedia)

A FEW MORE TRICKS

- Distance function iteration count is added to the color. This makes the structure glow. Also it acts as a diffuse+ambient light reducing light model complexity
- A fake normal for the surface is calculated by fudging x and y and normalizing this with constant z-offset. Combining this with front of the camera light we get very nice reflections
- Using clamping operations and math instead of conditionals in the iterators
- Replacing some operations with others f.e. length(x) -> sqrt(dot(x,x))
- No known GPU will run the scenery at 60 fps. We needed to overstep heavily and this made many structures buggy. We fixed this issue by limiting camera views to those that show little of buggy scenery only
- And most importantly we chose scenery that do not immediately look like Mandelbox

GLSL CODE

```
float r,v,m,n=.1,g=1.,e=0.,d=gl_TexCoord[0].x*.1;
vec3 o=normalize(gl_FragCoord.xyz/720.-vec3(.9,.5,-.7)),s,p;

float t(vec3 g)
{
  for(m=1.,p=g,r=0.;r<14.;r+=1.,p-=2.*min(max(p,-1.),1.),
    v=2./min(max(dot(p,p),.25),1.),m=v*m+1.,p=g-v*p);
  return(sqrt(dot(p,p))-1.)/m;
}

void main()
{
  for(;n<9.&&g>.001;n+=g*1.6,e+=.01) g=t(s=o*n+
    (d<4.?vec3(d-7.,-2.9,5.3):d<8.?vec3(-5.3,-5.3,d-10.):
    vec3(4.3,4.9,d-8.)));
  gl_FragColor=vec4(1.,.3,0.,0.)*
    (pow(max(dot(reflect(vec3(0.,0.,1.),
    normalize(vec3(t(s+vec3(.001,0.,0.))-g,
    t(s+vec3(0.,.001,0.))-g,5e-05))),o),0.),50.)+e);
}
```

Init

Distance func.

Ray tracing

Camera views

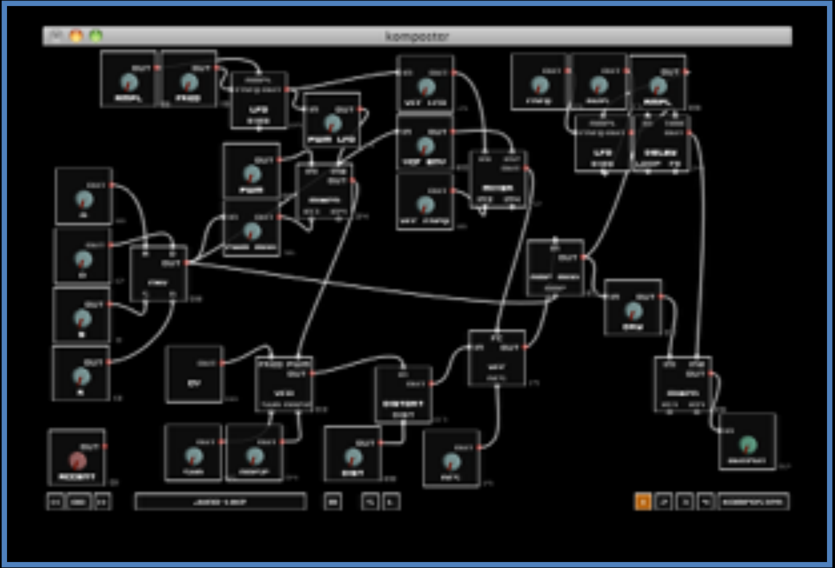
Lighting model

AUDIO

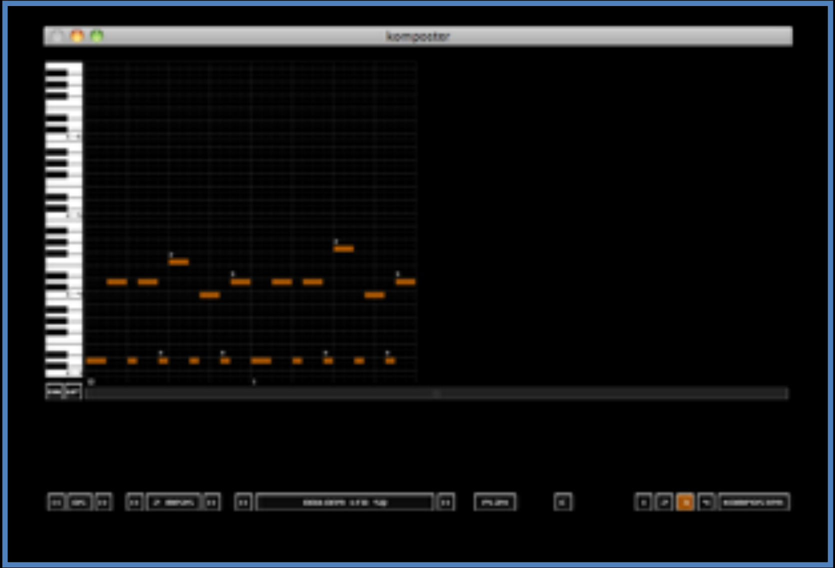
SOFTWARE SYNTHESIS

Modern 4 kilobyte intros typically contain a highly optimized software synthesizer with multiple oscillators per voice, biquad filters and delay/phaser/chorus effects.

Custom-built tools are used for creating instrument patches and composing.



Komposter synth modules



Komposter sequencer

SOFTWARE SYNTHESIS

Our existing software synthesizer was too large for use in a 1K intro, even when cut down to a bare minimum subset of features.

A prototype synthesizer with only a single oscillator, ADSR envelope and a simple IIR filter was tested but adding the actual music data and sequencing code would've bumped it too far over our "budget".

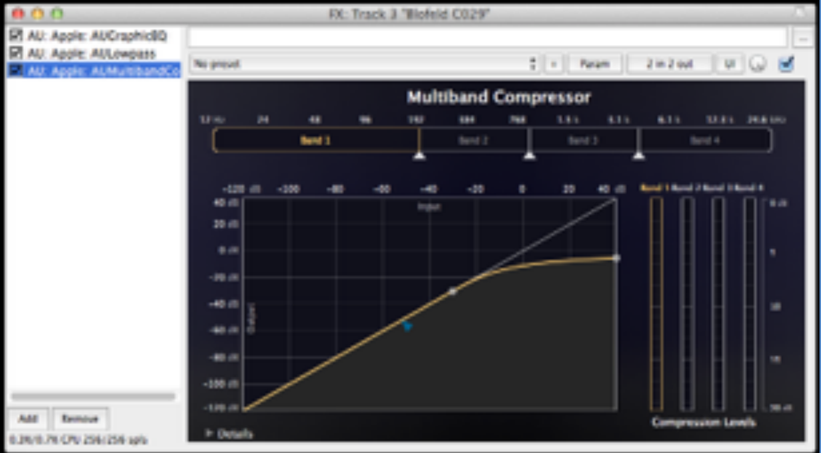
Further cutting of features would limit the soundtrack to just a monotone droning sound instead of music - not what we want.

AUDIO UNITS

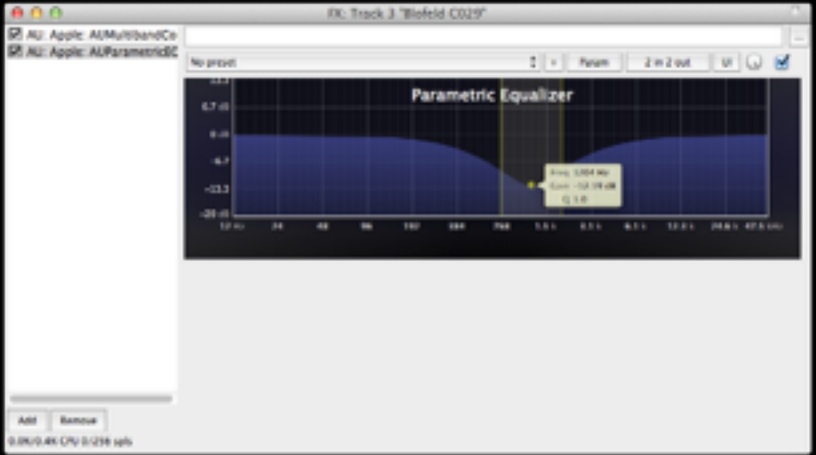
Mac OS X comes standard with a number "audio units" similar to VST plugins. They expose a clean C-language Carbon API to client applications.

In addition to effect units (delay, high/low/bandpass filters, reverb, etc), a General MIDI and Roland GS –compatible wavetable synthesizer (DLSSynth) and patch set is included with the OS.

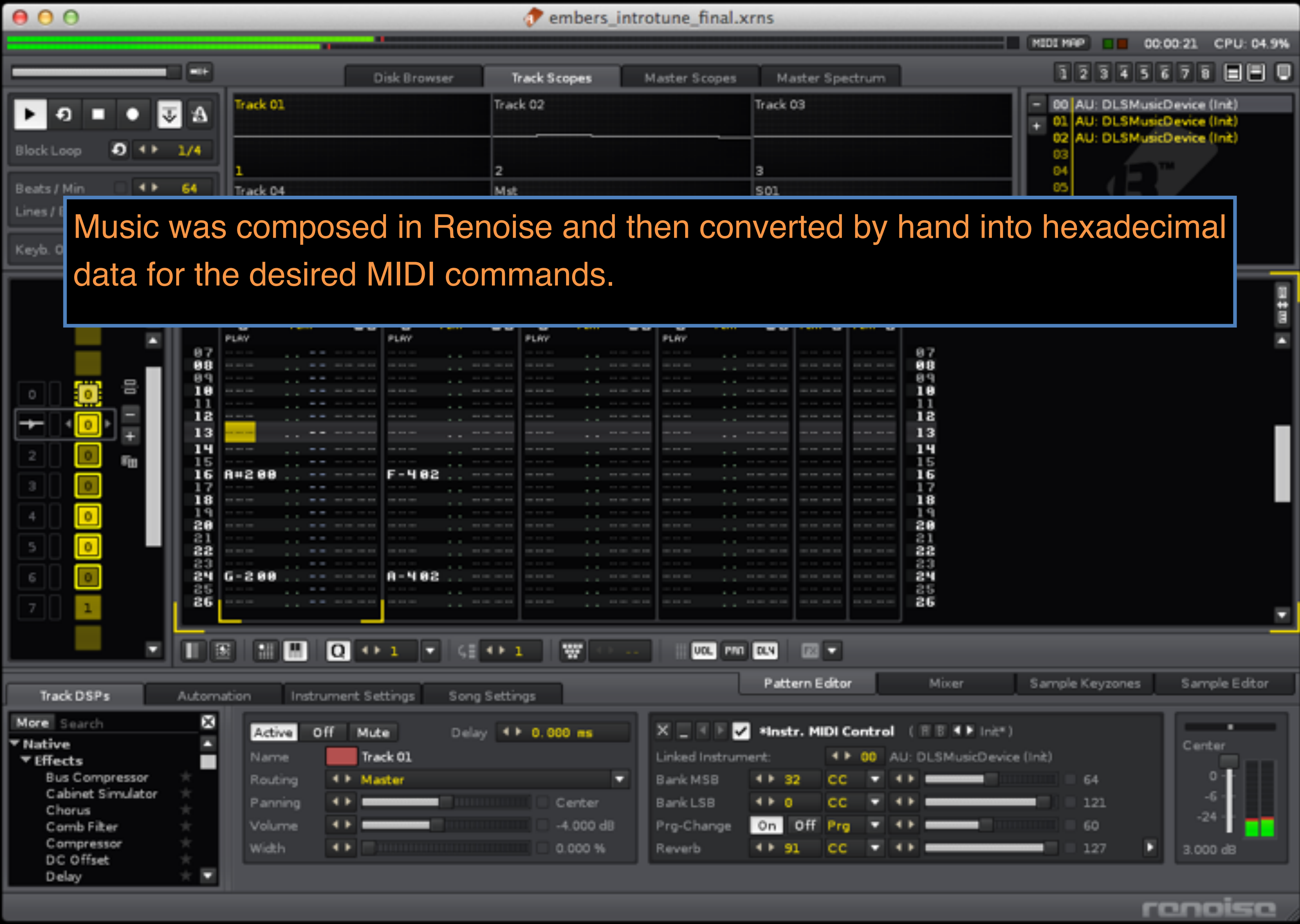
DLSSynth can be controlled using the MIDI command protocol.



Apple AU compressor



Apple AU equalizer


Music was composed in Renoise and then converted by hand into hexadecimal data for the desired MIDI commands.

embers_introtune_final.xrns

00:00:21 CPU: 04.9%

Block Loop 1/4

Beats / Min 64

Track 01	Track 02
16 A#2 00	F-4 02
24 G-2 00	A-4 02

00 AU: DLSMusicDevice (Init)
01 AU: DLSMusicDevice (Init)
02 AU: DLSMusicDevice (Init)

Name: Track 01
Routing: Master
Panning: Center
Volume: -4.000 dB
Width: 0.000 %
Delay: 0.000 ms

*Instr. MIDI Control
Linked Instrument: 00 AU: DLSMusicDevice (Init)

Bank MSB	32	CC	64
Bank LSB	0	CC	121
Prg-Change	On	Prg	60
Reverb	91	CC	127

MIDI

In uncompressed form, the music data alone was 372 bytes.

Examples of MIDI commands used:

```
0x80, 0x32, 0x7f ; note on, channel 0, note 32 (D on 4th octave)
                 ; velocity 127

0x70, 0x32, 0x7f ; note off, channel 0, note 32 (D on 4th octave)
             ; velocity 127

0xc1, 0x3c       ; program change on channel 1, program 3c

0xb1, 0x91, 0x7f ; controller change on channel 1, controller 91
             ; (reverb), value 127
```

OPTIMIZING

To ensure good compressibility, the music data had to be optimized:

- only 0 and 127 were used as velocities and controller values
- music was transposed to get MIDI note numbers which coincide with ASCII character codes
- instead of looping patterns, MIDI data was "unrolled”
- music playback position is also used as an animation counter

AUDIO CODE

```
MusicPlayer player; MusicSequence sequence; MusicTrack track;
NewMusicPlayer(&player);
NewMusicSequence(&sequence);
MusicPlayerSetSequence(player, sequence);
MusicSequenceNewTrack(sequence, &track);
```

Init

```
int i;for (i=0;i<3;i++)
  MusicTrackNewMIDIChannelEvent(track, 0, &music_init[i]);
```

Instrume
nts

```
i=0;float f=0;
while (music_data[i*2+0]) {
  MusicTrackNewMIDINoteEvent(track, f, &music_data[i*2+0]);
  MusicTrackNewMIDINoteEvent(track, f, &music_data[i*2+1]);
  f+=music_data[i*2+1].duration;
}
```

Note
events

```
MusicPlayerStart(player);
```

Start
playback

```
MusicPlayerGetTime(player, &ticker);
```

Get ticker in main
loop

MUSIC RESULTS

A total of about 200 of the 1020 bytes used for the soundtrack:

- 8 CoreAudio API functions referenced: ~30 bytes
- code for initializing the synth and playing the music: ~40 bytes
- music data: ~130 bytes

THE END

```
const int main[]={195};
```