›THE MAKING OF CANDYSTALL‹

OR HOW TO WIN ASSEMBLY USING A BUCKETLOAD OF CUBES

BY BLUEBERRY OF LOONIES, PACKMAN OF PITTSBURGH STALLERS, THE DANISH MUSICIAN OF PITTSBURGH STALLERS AND LEMMUS OF LOONIES

This is the story of Candystall, the winner of the Assembly 2007 4k intro competition. In this article, we describe the ideas that eventually led to the intro as it came to be, and the tools we created along the way to assist in the process.

THE OBJECT ENGINE

The object generation method we used in this intro was motivated by a desire to create objects that were more dynamic than the (nice, but somewhat boring) static meshes we used in Benitoite, our previous intro. No fancy meshes this time (been there, done that, got the T-shirt), just do as everybody else does in 4k: throw together lots of primitive objects (cubes, spheres and the like) to form interesting shapes and let them move in interesting ways. And of course, thinking like 4k coders, we wanted to create a single formalism with which we could express lots of different objects.

We were very much inspired by the seminar about procedural 3D object generation, given by Tomkh at Breakpoint 2006. The seminar described a method for building objects out of spheres. The fascinating thing about the method, we thought, was that it allowed the object definition to be recursive. Different parts of the object could refer to the same sub-object, and by referring to a sub-object within the sub-object itself, repetitious and tree-like structures could be obtained, leading to some quite interesting formations with very small definitions.

“WE WERE VERY MUCH INSPIRED BY TOMKH’S SEMINAR ABOUT PROCEDURAL 3D OBJECT GENERATION.”

The mesh calculations needed to transform a collection of intersecting spheres into a single object are probably too complex to have any use in 4k intros (until someone actually does it, of course). But the method of putting together sub-objects using recursive transformations appealed to us. We wanted to have some sort of the same overall capability for structuring an object, but instead of trying to merge everything into one mesh, we would just make the object consist of geometric primitives. An object generated by the formalism would simply be a list of primitives, each with its own position, rotation and scaling.

The method we ended up with was this: put transformations in a tree and have primitive objects in the leaves of the tree. Each transformation affects everything in the sub-tree below it. To express repetition, we can have loops in the tree (so it’s no longer a tree, strictly speaking, but we like to think of it as a tree). The loops are labeled with the number of times the loop should be taken. A loop below another loop gives two-dimensional repetition. Two loops from different branches of the tree produce fractal-like structures.

This formalism still only creates static objects. To make the objects dynamic, we extended the tree structure with expressions and variables. Each parameter to a transformation, and each color component for the color of a primitive (warning: this is the recipe for coder colors), is an expression which is evaluated as the tree is traversed. The expressions can contain a small selection of useful operations and functions (add, subtract, multiply, divide, sin, exp, clamp and round), plus variable references and constants. Furthermore, there is a special tree node for as-

signing the result of an expression to a variable. All nodes below the assignment see the new value of the assignment. When the traversal returns through the assignment node, the old value is restored. As it turns out, variables and expressions are useful for much more than just animation. By changing the value of a variable in a loop, the object can be shaped in more interesting ways than would be possible with just repeated transformations.

Everything is controlled by the predefined variable “time”, which simply contains the current time position in the intro, measured in music ticks. Further, for every instrument in the music, there is a special instrument variable which contains the amount of time elapsed since that instrument was last played. This makes it very easy to make the effect react to the music.

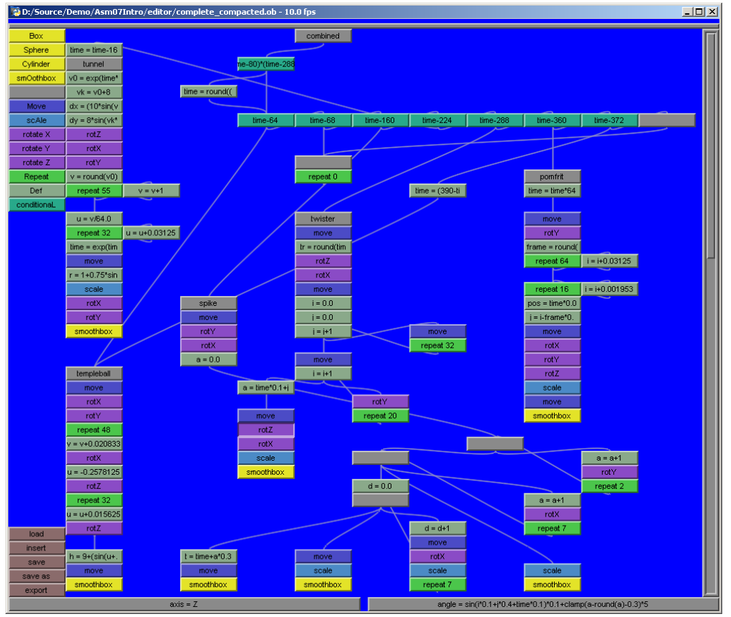
Finally, we needed a mechanism for actually changing the composition of the object over time, rather than just distorting it. For this, we introduced conditional nodes that select between two subtrees depending on the value of an expression. These nodes are also ideal for controlling the global switching between scenes in the intro; the whole intro is just one object that changes appearance depending on the time variable. “If time is less than 32 then render this effect, otherwise if time is less than 64 then render that effect, otherwise...”

THE TOOL

We knew from the start, that in order to get decent results out of this object generation engine, we would need some kind of editor with immediate visual feedback. Something where you could pull a slider for any parameter and see the result while pulling. And where the object tree could be manipulated graphically. Somewhat inspired by .werkkzeug, we went for a “grid of boxes” design. You create boxes for the various kinds of nodes you can have in the tree and connect the boxes with lines. Simple and efficient.

At the push of the button, the editor then serializes the whole thing into a data file that can be included in the intro and interpreted by the object generation code. The expressions defining the parameters (which are entered in textual form in the editor) are parsed and converted into a pre-order form (or polish notation, that is, the reverse of reverse polish notation) which is compact and easy to interpret.

A full-featured editor like this is a bit of a project in itself, and we needed to throw it together really quickly, so we wanted to make it as easy for us as we could. Just as the tool itself was essential for creating the intro, it was essential to use the right tool for creating the tool. We took this as an opportunity to learn Python. This is supposed to be a language which is good for throwing things

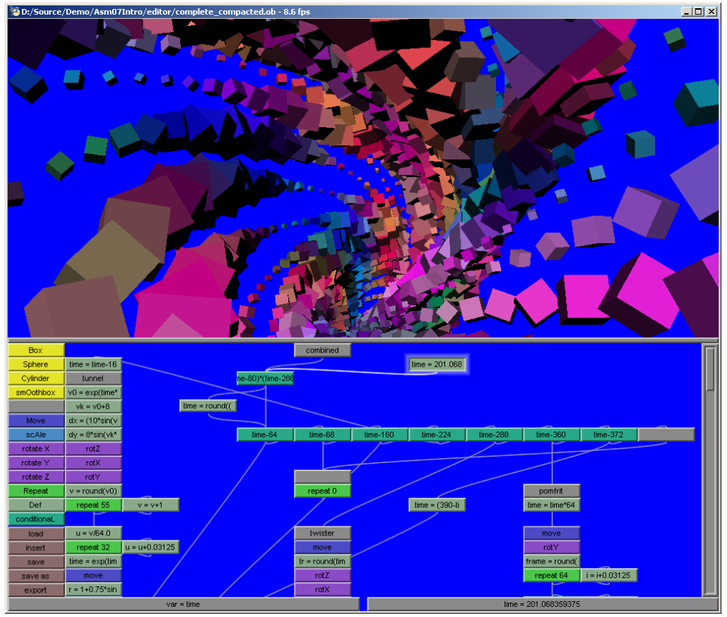


A screenshot of the tool with the object definition for the entire Candystall intro

together easily, and we must say, it lives up to its promise. Python is of course “a bit on the slow side”, but the tool was effective nonetheless. All in all, the tool consists of around 2000 lines of Python and uses the DirectPython wrapper for Direct3D. In the first, quickly-thrown-together version of the tool, all the tree traversal and calculation of expressions and transformation matrices were done in Python, with each primitive rendered by a separate draw call. This was, of course, painfully slow. We have since rectified this so that the tool now calls into a custom DLL containing essentially the object generation engine from the intro. Now it is actually the GUI which is slowing things down.

USING THE TOOL

The flexible nature of the formalism allowed for some interesting ways of controlling the animations of the effects. Since all timing is done by reading the time variable, and the time variable can be altered just as any other variable, the scripting of the effects could be separated from the effects themselves.



Effect preview inside the tool

For instance, we got the idea that the tunnel should speed up instead of running at constant speed. This was simply done by inserting a a “time = exp(time*.0625)*16” assignment into the tunnel effect. Similarly, the very last temple ball reprise was done by negating the time variable and calling the first effect again, thereby running the effect in reverse.

The strobo effect at the start of the “spider” and “pommes-worm” effects was also obtained by altering the time variable. At the very top of the tree - before the effect selecting branches - the time variable is conditionally rounded up to sync to the music. The condition is a polynomial in the time variable which is negative during the periods where the stroboing should take place. Hence, the stroboing effects do not know that they are stroboing; the time variable just happens to be constant, thereby rendering the effects the same way for some frames.

Naturally, we quickly hit the limitations of the formalism. For instance, since the number of repetitions for a loop is stored in a single byte, it is not possible to repeat a loop more than 255 times. If you need a loop of greater depth, you have to do two nested loops. Unfortunately, this does not exactly do the right thing. Looking at the tree structure, a 2048-repeat can be thought of as expanding into a tree of depth 2048. But if you implement the 2048-repeat as a 128-repeat containing a 16-repeat, then you get a tree of depth 128, each level containing a sub-tree of depth 16. Since state changes are undone when the tree is traversed upwards, this means that, in the outer loop, you need to manually redo all changes done in the inner loop. E.g. if the inner loop does an “i = i+1”, then you need to also do “i = i+16” in the outer loop to redo the effects of the inner loop. This, of course, means repeating code, and repeating code is a Bad Thing - especially in 4k.

“REPEATING CODE IS A BAD THING - ESPECIALLY IN 4K.”

We had other examples of needing to repeat code. The expressions defining the shape of the tunnel are repeated three times in the tree. Once in order to draw the tunnel, once in order to position the camera in the middle of the tunnel, and once to make the camera look in the right direction - left when the tunnel goes left, etc. Because the expressions are identical, Crinkler eats them more or less away, of course, but it was a source of error that you had to do the same change in several places when changing the shape of the tunnel.

The fact that we interpreted the tree in Python code also caused problems besides those of performance. As Python logic differs a bit from assembly logic, we occasionally saw some differences between how the effects looked in the tool and in the intro. For example, our Python implementation of round() rounded numbers

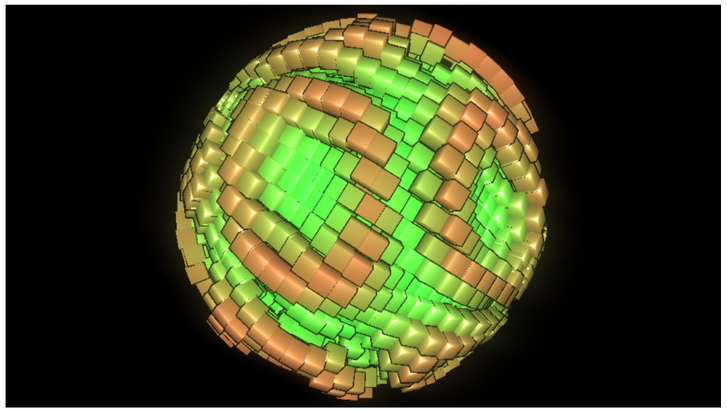
ending in .5 upwards, whereas the x86 FRNDINT instruction uses banker's rounding, to the closest even number. This caused us quite some panic close to the deadline.

RENDERING

The original idea for the intro was to combine the generated objects with a volumetric light effect. However, about a week before Assembly we realized that the method was not really that well suited to modeling scenes appropriate for this effect, especially not in the time we had left, nor with the then experimental state of the tool and object generation. Thus, we postponed that idea for some other time, and decided to throw something quickly together instead. Mainly to get some experience playing around with the tool.

"WE BRIEFLY CONSIDERED CALLING THE INTRO ROYAL TEMPLE STALL."

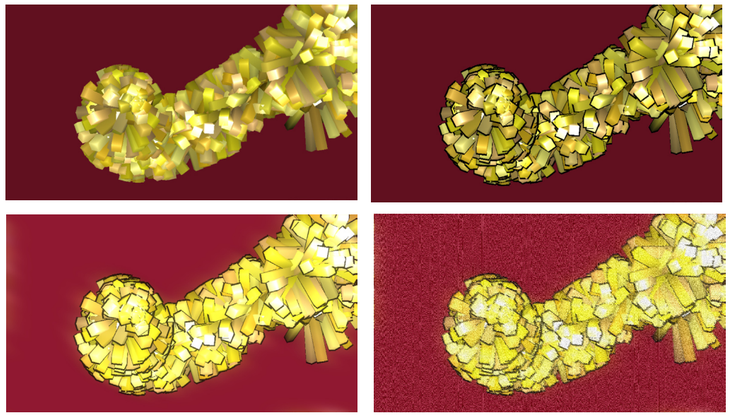
It so happened that we came to look upon sts-05: Royal Temple Ball, and we liked what we saw, and thought: "Hey, let's do that." We briefly considered calling the intro "Royal Temple Stall", but it wouldn't have been fitting, since the intro is after all not really a remake or parody as such, just somewhat inspired.



The opening scene of Candystall, "somewhat inspired" by sts-05: Royal Temple Ball

The cel-shading effect is of the really simple kind - draw everything in black with the vertices offset a little along the vertex normal and backface culling the other way 'round. This produces lots of weird artifacts, but it was Good Enough (tm) for us. The "candy shader" which we used to shade the boxes themselves came about more or less by chance. It is simply an ordinary per-pixel lighting shader with the normalization removed. It had this magical ability to make our coder colors look nice and candy-like.

Nowadays it is almost mandatory, even in 4k intros, to have some kind of post-processing to beef up the visuals. Everybody needs glow, of course. And some distortion and noise to put on top is always nice (or maybe it isn't). The strength of the individual post-processing effects is controlled by a script so they can vary throughout the intro and react to the music (hypnoglow for the masses!)



The same effect with varying levels of rendering complexity: Plain candy-shading, with outline, with glow and with distortion and noise.

IMPLEMENTATION - THE GRITTY DETAILS

The intro code consists of roughly 2000 lines of asm and 100 lines of HLSL. Some parts of the code, such as the object graph interpreter and the synth, were written in asm from the start, since these are the sort of tasks that asm is good for. Most of the rendering and control code was written in C++ for easier experimentation during development and then later rewritten in asm. It is somewhat cumbersome to do so (though not really boring - we are coders, after all), but it had to be rewritten to make it small enough. The code produced from C++, or at least from the Visual Studio compiler, is often horribly big (not to mention very packer-

unfriendly) even for simple things like calling a COM function. The code that comes out of the compiler probably runs faster than the asm we produce by hand, mind you, but that is a somewhat weak selling point for 4k intros. Most of the code (especially the D3D interfacing code) isn't really time-critical anyway. Writing everything in asm also allows us to keep the coding style consistent, which further helps the packer.

A new thing we tried (well, new for us, anyway) was to use the DirectX FX file format for managing all the shaders used in the intro. This turned out to work really well and saved us a lot of shader handling code. This interface is quite easy to use. The process goes like this: Write all your shaders in a single text file and define a number of rendering passes consisting of vertex/pixel shader pairs. Compiling all the shaders is now just a single function call (D3DXCreateEffect), and the shader pairs can then be referred to simply by their number, using a pair of method calls (BeginPass/EndPass) on the effect object. There is a lot more you can do with FX files, such as setting various render states, but most of these can probably be done more compactly using normal D3D calls.

The FX file gets embedded into the intro as text. Text data packs quite well as it is, but it can of course be made even more compact by stripping whitespace and comments, choosing short variable names, inlining variables and functions only used in one place and so on. Apart from being a quite laborious task, doing this to your shader has the drawback that it renders the code quite unreadable, and even less editable. In order to keep our shader source human-readable during development, and to be able to make changes to it late in the process (which we did), we wrote a small tool (also in Python) to perform this "obfuscation" process for us automatically.

"LETTING TONE DEAF CODERS TAKE CONTROL OVER THE DEVELOPMENT OF A SYNTHESIZER IS A DEFINITE RECIPE FOR DISASTER."

For each frame, the object generator simply traverses the object definition tree recursively, accumulating transformation matrices and variable contents along the way. When it reaches a primitive (in the case of Candystall, all primitives are cubes), it transforms the primitive using the current transformation matrix and writes the result into a dynamic vertex buffer. Thus, all vertex transformation is done on the CPU. As usual, there are a couple of D3DX functions that come in handy for doing tasks like this - the D3DXVec3TransformCoordArray and D3DXVec3TransformNormalArray optimized transformation functions are used to do the actual transformation, and the D3DX-supplied matrix stack is used to calculate the transformation matrices and to keep track of intermediate transformations during traversal.

The space taken up by the various parts of the intro are roughly

	uncompressed	compressed
Startup and init code	600	250
Rendering, postprocessing	900	350
Object gen, mesh handling	700	350
Synth	750	500
Object/animation data	1500	800
Music data	3400	800
Shaders	1300	400
Postproc. scripting data	300	50
Misc		100
Overhead		450

THE SYNTHESIZER - HOW TO ROCK THE ASSEMBLY SOUND SYSTEM

It is common knowledge these days that 4k intros need music and good 4ks need good music. In Benitoite, we relied on the kklangzeug synthesizer, which was kindly provided to us by Gargaj and tsw of Ümlaut design. Huge respect for lending us their tools in a time of need.

kklangzeug comes with all you would need to get music in your 4k, a full-fledged tracker-like GUI and a small replay routine. This was the easy way out which we needed at the time. Without kklangzeug it would have been impossible for us to finish in time for the Breakpoint '06 deadline. The observant reader will notice that we didn't actually make the deadline anyway, but that is besides the point. Although we missed the deadline, we decided to stick to kklangzeug.

"DELAY IS A MUST-HAVE IN 4K INTRO MUSIC!"

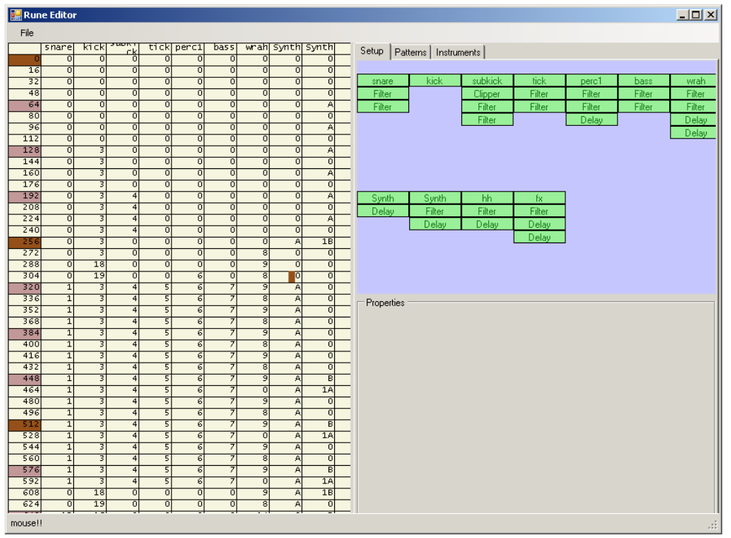
Despite having contributed to kklangzeug with numerous optimizations and tweaks, we found that it was time for us to move on to our own synthesizer. We did this partly as a learning experience and partly as a chance to design a new synth from scratch. We could of course just rely on midi abuse for music, which seems to be popular these days, but we really wanted to do our own thing. Candystall was the first intro to use this synthesizer. It had strictly speaking already been used in our contribution to the executable music compo at Solskogen, but probably few people noticed.

To have any chance of success, we teamed up with the two musicians Toxic Child/Mainloop and Puryx/TBC, since coders, as most people know, generally have no clue about composing music. Letting tone deaf coders take control over the development of a synthesizer is a definite recipe for disaster, as can be witnessed in quite a few 4k intros. We relied on feedback from our musicians to keep the size-optimizing demons within us from making an unusable, but extremely tiny, synthesizer.

Based on our previous experience with kklangzeug (and its limitations) it was clear to us from the beginning that support for stereo and a dynamic filter cutoff were must-have features. And after our musicians had been spoiled with the GUI of kklangzeug, there was, unfortunately, no way around it: we had to design a user interface as well.

The GUI was written in C# using the Windows Forms framework. To avoid rewriting the synthesizer in C# we exported the actual replay code to a DLL, which could then be called using unmanaged/managed interop. Some parts of the synth, of course, still needed to be specialized into separate versions for the GUI and for the actual intro, but this was easily handled using what eventually turned out to be excessive amounts of conditional compilation. The primary difference between the two is that the intro replay code generates the entire song in a single pass, whereas the GUI player is intended to be able to generate any part of the song in realtime.

The GUI is tracker based and composing is thus done by creating patterns of 16 notes and then playing these in succession on a number of instrument machines (the topmost boxes in the GUI). The output of each of these instruments is then propagated through a number of other machines, such as delays, filters, etc., until the final signals reach the bottom, where all channels are mixed into one signal. The setup of these machines is not fixed, and it thus enables for many different types of sounds and musical styles.



The synthesizer GUI with the Candystall music - Sequences of patterns on the left and synthesizer setup on the top right.

The machines(boxes) are the core elements of the synth. In order to customize the behavior of each instance of the machine they each have a number of parameters associated with them. Param-

eters are mostly 32-bit floating point numbers, except a select few, which are 32-bit integers. There is a quantization slider for each property controlling the number of significant bits needed for this specific parameter. We found it very convenient to let the musician control the quantization, as he knows which parameters need the extra precision.

This approach can immediately give larger data than, for instance, using a fixed quantization to single bytes. But after proper quantization and crinkling, we found this to be nearly as compact, as Crinkler easily recognises the zeroes in between significant bytes. This has the added benefit that we don't need to do any unpacking or rescaling of parameters in the synth, which makes the replay code even smaller.

Finding the right trade-off between features and size is the real challenge. To keep the code small and simple, all calculations and intermediate buffers are kept in floating point and full stereo. This does increase the requirements a bit, both in terms of memory usage and cycles, but for 4k coding this is acceptable. Like most 4k synthesizers, we don't do anything actively to avoid aliasing artifacts. We could, without great hassle, sacrifice some cycles, and bytes, to do some oversampling, but we found that the artifacts were not severe enough for us to bother, at least not in 4k.

We found, together with our musical advisors, that the following set of machines formed a good compromise.

- Instrument (tone generator).
This is the only machine actually generating sound. It plays waveforms based on a given sequence of patterns; the frequency of the waveform can remain constant throughout the note or slide either linearly or exponentially to another frequency. The synthesizer supports three types of oscillators: saws, squares, and sines. So far this makes for some pretty thin and uninteresting sounds.

To add more depth and variation to the sound, a total of three such oscillators can be combined, each with separate phase offset and symmetric detuning. The detuning is performed by adding two oscillators to the waveform, which play frequencies equally spaced above and below the main frequency. The three waveforms are combined using any binary floating point instruction. Popular choices are add and mul instructions. The signal is then mixed with white noise and the volume of the final signal is modulated with a simple ADSR-envelope. Although these parts are simple, they represent the most bulky piece of code in the synthesizer.

- Delay.
The delay, as most 4k coders know, is a great tool for turning simple bleeps into wonderful atmospheric sounds. Given its low complexity, we saw no reason not to include it in the synth as well. The implementation is a basic feedback delay which crosses the left and right channels of the signal and thus effectively flips the channels for every delay length. This gives a nice echo effect of off-centered sounds.

"WHAT DOES THE CROWD WANT? BASS!"

- Filter.
The filter supports low-, high- and band-pass filtering of the input. The cutoff frequency has two hard-coded sinusoidal modulators, each with separate frequency and amplitude. This gives some control over the progression without adding too much extra data or code complexity. It still gives the impression of a much more dynamic sound compared to a constant cutoff.

- Compressor.
The Compressor scales samples with magnitude exceeding a given threshold by a constant fraction. Compression or limiting is often used to play more instruments at the same time, without some of them squelching the others. Compression is usually done based on some sort of running average of a window of

preceding samples. As this is 4k coding, we use a much more crude approach. This technique actually originates directly from kklangzeug, but one of our musicians insisted that we needed this as well. One could argue that this approach is more of a distortion effect. One particular nice application of the compressor is that it can be used for clipping by using a scaling factor of 0.

“WE WERE SOMEWHAT CONCERNED THAT PEOPLE WOULD NOT LIKE OUR HAPPY CODER COLORS.”

This is basically all it takes to generate the music. The synthesizer is written entirely in assembly language and the resulting code compresses to between 500 and 600 bytes depending on the features needed for the track. For actual playback we went with the WaveOut API, as we found that it was smaller than using DirectSound API and has more reliable timing features than PlaySound, which offers no means to sync the music to the visuals, at all.

MUSIC

So, now we have this nice synth - but could we also manage to create a great 4k tune? Of course!

Since the concept for our Assembly 4k changed along the way, we went through a couple of different musical genres, before finally settling on that the music should probably be ordinary and appealing to a crowd.

Blueberry also requested happiness in the music, and thus The Danish Musician had to come up with a lot of melodies, which is usually not his domain. The result was the track for Candystall: a slow paced, melodic trance-like track with many breaks, and nice stereo effects.

Usually, our workflow when doing 4k music is that some melodic experiments are done in normal sequencing software, like Reason. It is much easier to play around with melodies and delay settings in a normal sequencer, before moving into our homemade music GUI. When a suitable melody has been created, it can be edited into patterns in the editor GUI, and the real fun begins.

The main melody is a synth with two oscillators: 1 sine and 1 saw, together with a tiny bit of detuning and phase offset. Of course, the instrument also has a bit of stereo panning and delay - delay is a must-have in 4k intro music!

When a suitable instrument has been built to match the main melody, we need the aforementioned crowd-appealing quality. What does the crowd want? BASS! Luckily, the synth was constructed with this in mind - and thus we really do have the ability to tweak out some awesome kick drums. A long, deep, kick drum is of course one of the main ingredients in the music for Candystall.

OK, the basic melody and kick drum were now quite settled, so we needed an intro part to the tune - some cool sound that builds up a bit of tension. Stereo sounds are good at that, and luckily the stereo setting in the synth, combined with a very short delay, and then a longer delay afterwards, creates a really nice wide effect on the filtered square that you hear in the very beginning of the tune.

Some high-pass-filtered white noise waveforms serve as hihats, and bandpass filtered white noise serves as a snare. The white noise percussion is probably the weakest part of our synth, but hey, you can’t have everything in 4k.

Finally, there’s the electro-like sound at the beginning and end of the track, which is 2 square waves offset and detuned like hell. We hit a lucky parameter, creating a great sound. The tune has 11 different tracks, and accordingly 11 different instruments with various effects applied.

Add a some sound effects for the final part of the tune, some LFO-controlled filters to create some variation in the main intruments along the way, and we basically have the Candystall tune, which you know by now.

THE INTRO COMING INTO EXISTENCE

As is probably apparent by now, though some of the the effects are obviously heavily inspired by sts-05: Royal Temple Ball, there wasn't much of a concept behind the intro as a whole. It was merely the result of playing around with our newly created tool and throwing the resulting effects together into an intro. The result is naturally quite incoherent, but at least there is a lot of content.

This approach of just throwing something together lessened our ambitions somewhat, probably giving us a much more relaxed attitude towards the whole project than we would have had if the intention had been to show off our wonderful new object generator in all its glory. It has been refreshing to do something truly experimental.

We were somewhat concerned that people would not like our happy coder colors, but as it turns out, we have received far more positive than negative comments on the colors. That was a pleasant surprise; probably the candy shader doing its magic.

The music and effects were developed largely independently of each other, so it was a bit of a lucky strike that they actually fitted so well together. When the three of us who were at Assembly received the freshly created music from The Danish Musician, just a few days before the deadline, we all liked it, and the whole flow of the intro fell in place naturally from the flow of the music.

GOING FORWARD

So, that was it - created the tools, used them, and now onto something completely different, right? Of course not. Candystall was just the initial experiment. Now that we have created these nice tools, we want to evolve them further and use them for more 4k intro projects.

We already have several ideas for making the formalism even more flexible. One very useful addition is global assignments whose effects are not undone when returning through the assignment. This makes it possible for different sub-trees to influence each other, which gives some fascinating possibilities. It also makes it straightforward to implement a kind of subroutine by simply attaching the subroutine as a branch and letting its effects affect the other branches. This can reduce code duplication considerably.

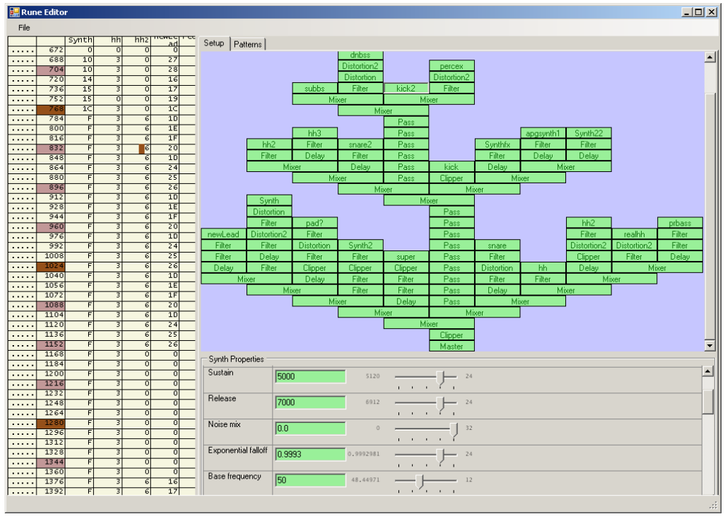
As it is now, a transformation always saves the old transformation state and restores it when coming back. If this state-saving is separated out into its own operations, transformations can also have effect across branches. We are experimenting with these features at the moment, and hopefully you will see the result before long.

"NOW THAT WE HAVE CREATED THESE NICE TOOLS, WE WANT TO EVOLVE THEM FURTHER AND USE THEM FOR MORE 4K INTRO PROJECTS."

As for the object editor, it did its job well, but it can of course be improved in many ways. First of all, we need to spend some time cleaning out some of the quirks that come from being thrown together quickly. Then we have a long to-do list of useful features, like splitting the object into multiple files, better keyboard control, more speed optimizations, copy/paste, etc.

Going forward, we would like to integrate even more of the intro design process into the tool, like having the music play while scripting, and using the actual intro shaders for the preview ren-

dering. With time, it could grow into a full-fledged intro creation tool along the lines of .werkkzeug. The question to ask ourselves is of course whether this is actually desirable. Only time will tell.



The new synthesizer GUI, showing the entire instrument setup for the three part track from minidisk.

The synthesizer has evolved somewhat since Candystall. The connectivity of the machines now forms a tree, with explicit mixer machines mixing tracks two at a time, rather than mixing everything in one go at the end. Each of the inputs to a mixer is weighted by a volume parameter. This allows for more flexible mixing of tracks and makes it possible to apply effects to multiple tracks after mixing. This is particularly useful for constructing more complex layered instruments by mixing simpler ones together.

The new synth also includes two distortion machines. They perform simple non-linear transformations of the signal, based on the trigonometric floating point instructions. The magic parameters of the transformations are exposed to the musician, so he can do his magic.

FINAL THOUGHTS

The construction and use of custom-made tools to create demos and intros has been fairly common practice for some years now. Tools for creating 4k intros have been made as well, but to the best of our knowledge, these have until now been mostly specialized for one particular aspect of the intro content - the music, the objects, compression etc.

“NEXT TIME WE MIGHT EVEN CONSIDER PUTTING SOMETHING A BIT MORE EXCITING THAN CUBES ON THE SCREEN.”

The Candystall project has been, more than anything else, an experiment in constructing and using a generic tool in which the entire composition and flow of the intro is modelled and scripted. It seems the experiment was a success, and we are looking forward to further developing the tool and using it for more productions. Next time, we hopefully will have the opportunity to spend more time on the actual design of the intro, and we might even consider putting something a bit more exciting than cubes on the screen.