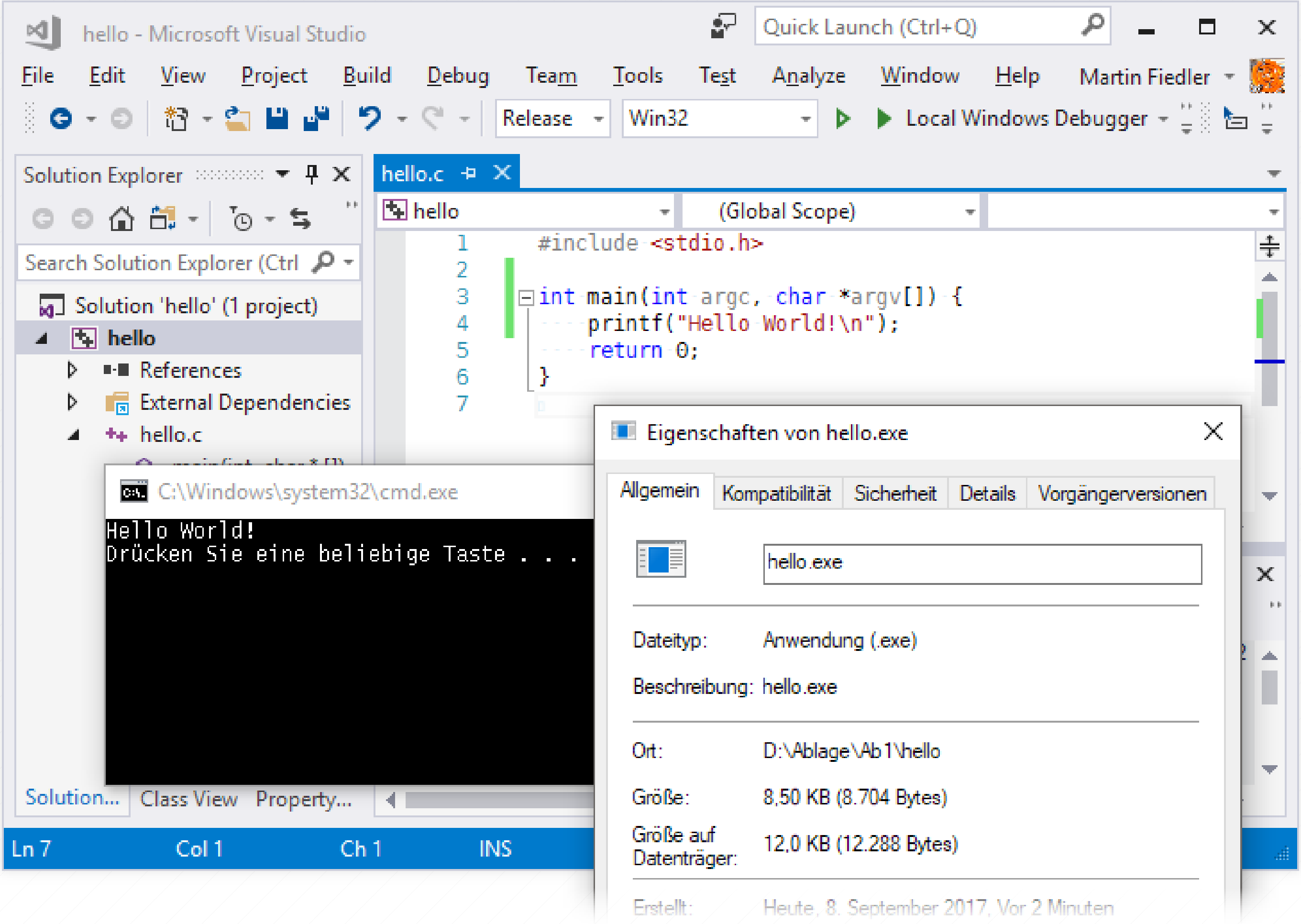
Martin Fiedler a.k.a.
KeyJ^TRBL

All The Small Things:
Tricks and Techniques used in Intros

Deadline 2017
September 30, 2017
ORWOhaus, Berlin, Germany

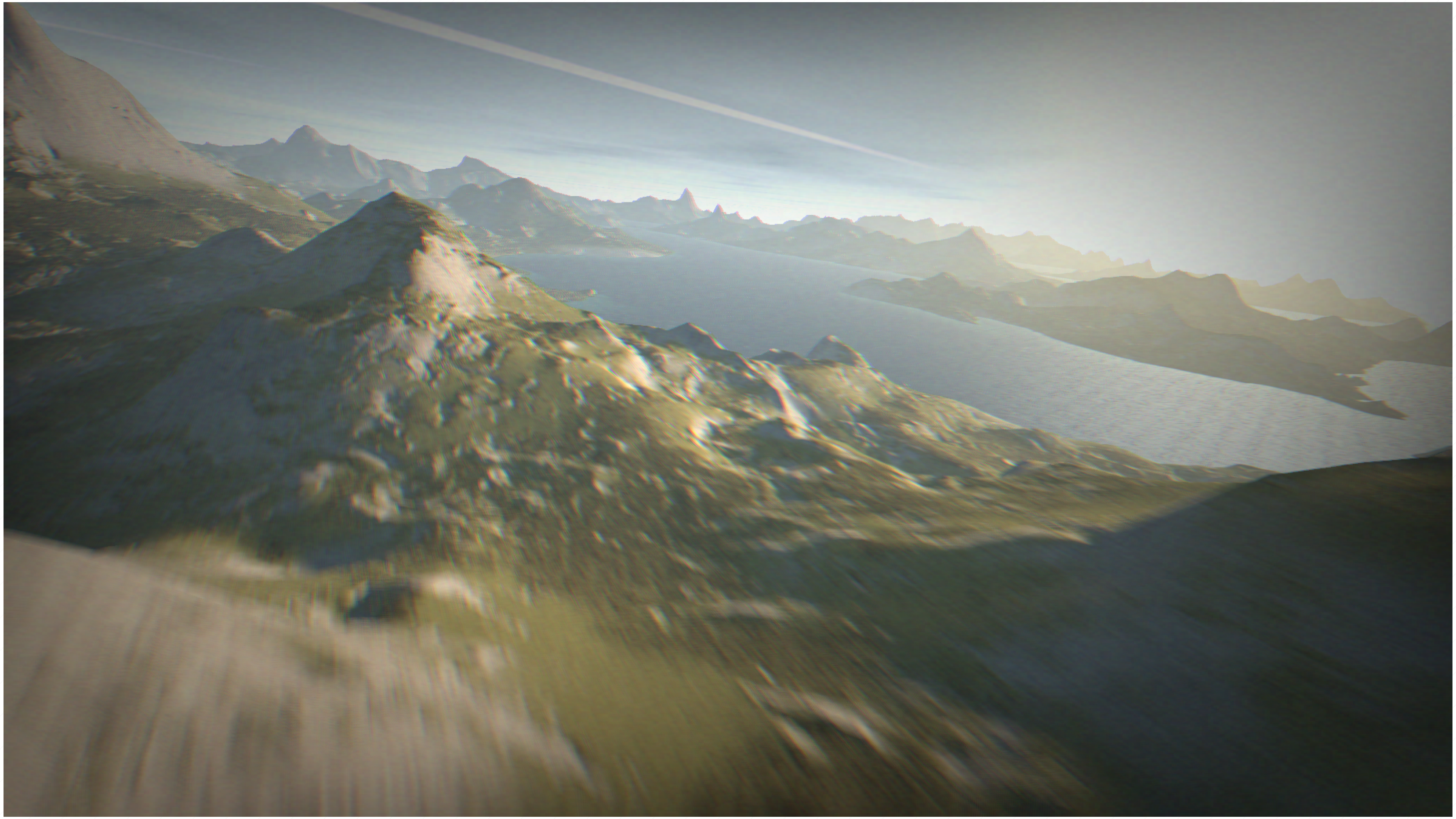
Intro(duction)



writes “Hello, World!”

8704 bytes

Intro(duction)



flyby over a beautiful landscape

3½ minutes of music

4066 bytes

Agenda

- general techniques ← *general audience*
  - procedural generation
  - demo-in-a-shader
  - saving code size
  - compression
- technical tricks ← *coder-oriented*
  - EXE header tricks
  - import by hash

Scope

- what to expect:
  - summary of some “state of the art” techniques used in intros
- what *not* to expect:
  - no original research
  - no revolutionary new method
  - no “how to make an intro” tutorial
- focused on the Windows platform
  - most concepts apply to other platforms, too

What does a demo consist of?

- Code *(code for the CPU)*
- Shaders *(code for the GPU)*
- Geometry *(“meshes” etc. – 3D object shapes)*
- Textures *(or image data in general)*
- Music
- Other Data *(e.g. animation control data)*

A survey of demo sizes

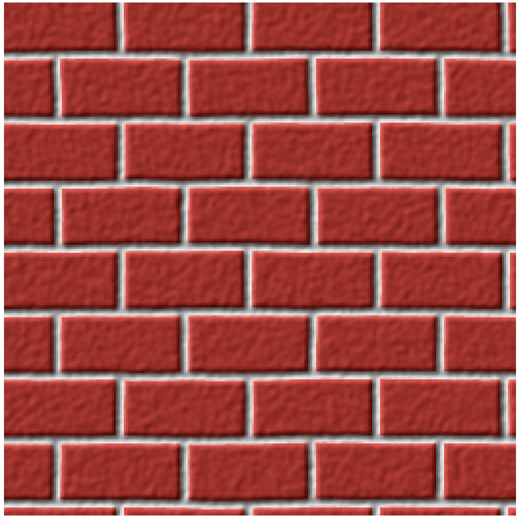
	Code	Shaders	Geometry	Textures	Music	Other
Lifeforce *ASD*	4.31 MiB	0.21 MiB	5.15 MiB	16.40 MiB	9.60 MiB	—
Stargazer *Orb & Andromeda*	1.46 MiB	0.66 MiB	22.85 MiB	37.29 MiB	4.22 MiB	1.11 MiB
1995 *Kewlers & MFX*	3.15 MiB	0.01 MiB	?	4.91 MiB	7.36 MiB	—
Agenda Circling Forth *Fairlight & CNCD*	6.78 MiB	0.01 MiB	169.90 MiB	23.94 MiB	9.19 MiB	1.68 MiB
Final Audition *Plastic*	0.89 MiB	?	5.94 MiB	5.66 MiB	3.44 MiB	0.33 MiB
Average *(without highest and lowest)*	~ 3 MiB *(8%)*	~ 0.1 MiB *(<1%)*	~ 11 MiB *(31%)*	~ 15 MiB *(42%)*	~ 6 MiB *(17%)*	~ 0.5 MiB *(1%)*

Procedural Generation

- textures, geometry data and music
  are typically the biggest parts of a demo
- idea: don't *store* them – *generate* them at runtime!
- only record the steps required to reconstruct the data
  - usually *much* smaller than the original data
- need to store the code that performs the steps as well
  - larger than the parameter data, but much smaller than the generated data
  - many parts can be re-used for multiple textures / meshes / synthesizers
- caveat: harder to use for the artists!
  - no Photoshop, no 3DStudio/Blender, no dozens of VST plugins ...

Procedural Textures

Example:
a simple brick texture
(256×256 pixels, RGB)

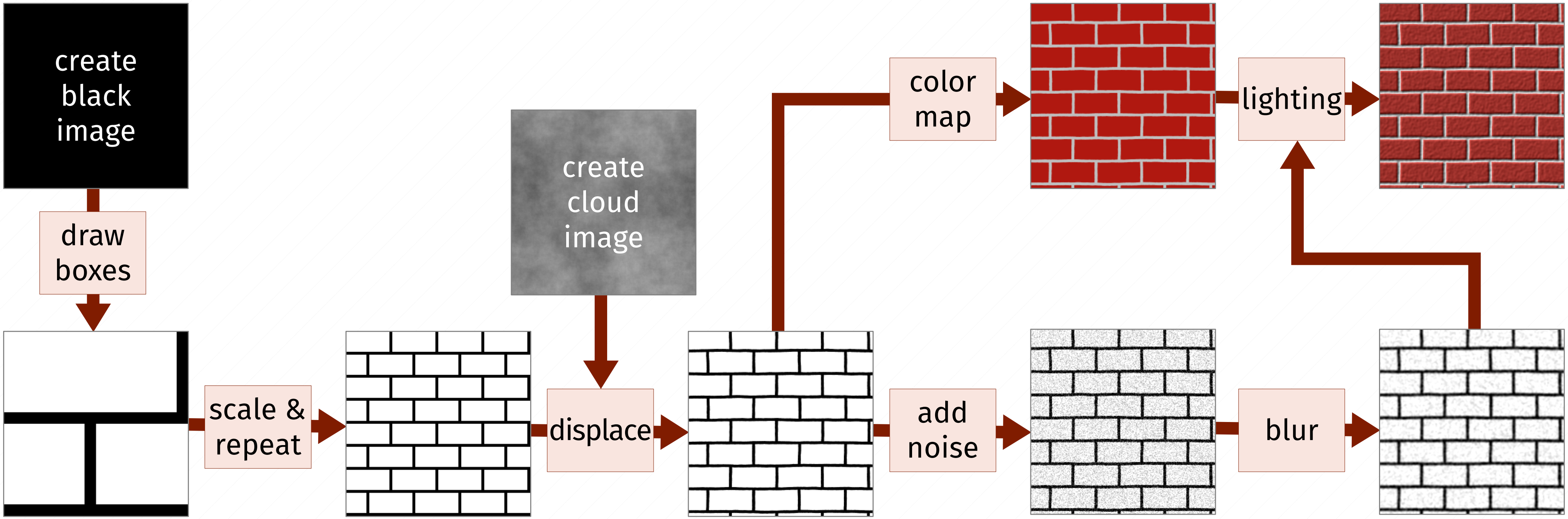


192 KiB uncompressed
~ 83 KiB PNG
~ 20 KiB JPEG

```
TGimage *A, *B, *C;
B = tgCreate(256,256, TG_FORMAT_GRAY | TG_FORMAT_WRAP);
tgFill(B, 0xFF000000);
tgRect(B, 0,0, 240,112, 0xFFFFFFFF);
tgRect(B, 0,128, 112,112, 0xFFFFFFFF);
tgRect(B, 128,128, 128,112, 0xFFFFFFFF);
tgRotoZoom(B, 0.00, 4.00);
C = tgCreateCompatible(B);
tgPlasma(C, 8192, 0.70);
tgDisplace(B, C, 7,7, 0);
tgFree(C);
A = tgCopy(B);
tgColorMap(A, 0xFFC0C0C0, 0xFFB01810);
tgNoise(B, 20480);
tgBlurEx(B, 2, 0.00);
tgLight(A, B, 0.12, 0.78, 0.78, 0.00, 1.00, 0.50, 2.00);
tgFree(A);
tgFree(B);
```

a few texture generator calls
< 200 bytes

Procedural Textures



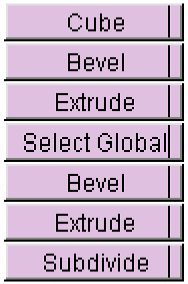
Procedural Geometry

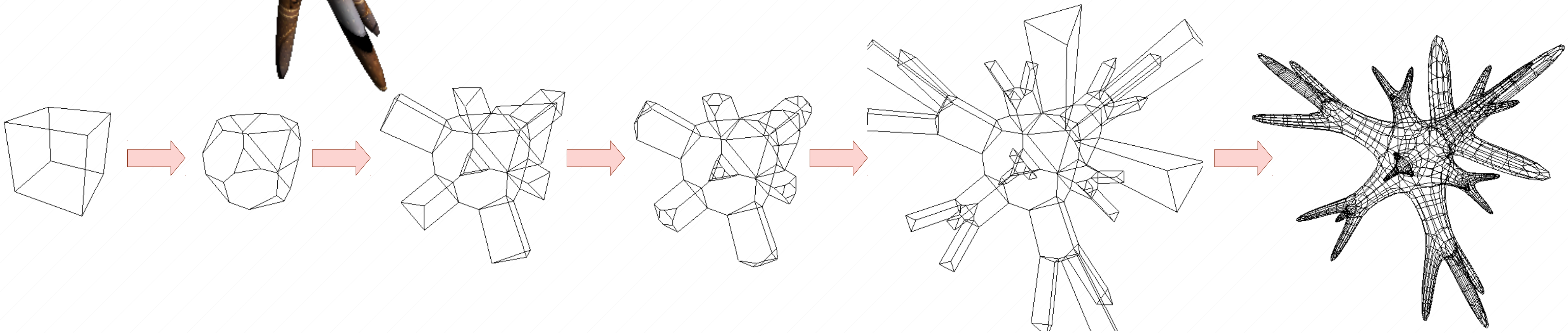
Example Mesh
(from the werkkzeug 1 tutorial)

2402 vertices, 4800 triangles

uncompressed: ~ 84 KiB

7 operators + parameters,
< 200 bytes





Procedural Textures and Geometry

- What about text?
  - no problem if you're OK with a standard Windows font!
  - GetGlyphOutline API produces small bitmaps or vector data for single characters ("glyphs")
- Code size of decent texture and geometry generators: 20-50 KiB uncompressed
  - but shared for *all* textures / meshes used in the intro!
- In 4k intros: typically "specialized" generators
  - code that generates *exactly* the desired texture / mesh; no control data
  - or use default meshes provided by the graphics API or commonly installed libraries (e.g. D3DXCreateBox, GLUquadric)

“Procedural Music” → Software Synthesizers

- for music, same approach can be used:
  - don’t play a ready-made track in MP3, Ogg or similar format
  - instead *synthesize* it in real-time
  - mostly oscillators and filters, little or no samples
- common “professional grade” demoscene softsynths:
  - 64k: V2, 64klang, WaveSabre, Tunefish
  - 4k: 4klang, Oidos, Clinkster
  - include a VSTi plugin for DAWs
    - musician composes a track using only this single plugin
    - notes and synth settings then exported into a compact format
- or a fully custom synth, entering notes as numbers in code :)

Why are shaders so small?

- recap: shaders are freakin' *tiny!*
- they are technically GPU code, but *not* machine code:
  - need to work with different GPUs with totally different architectures
  - either vendor-neutral *bytecode* (Direct3D, Vulkan)
  - … or actual source code! (OpenGL, Direct3D with d3dcompiler*.dll)
- source code in particular is very compact
  - lots of reasons …
  - most importantly: it compresses very well!
- Makes sense to do as much with shaders as possible!
  - Why not just render *everything* only with a shader?

Whole Demos in a Shader

- iq^rgba: “Rendering Worlds With Two Triangles” (2008)
- don’t use classic polygon rendering, but *raytracing* or variants
  - commonly *signed distance field ray-marching* a.k.a. *sphere tracing*
  - 2006-era GPUs became capable of doing that (Shader Model 3)
- a shader is run for *each* pixel of the screen independently
- geometry and textures are implicit
  - everything’s in the shader code
  - can use funky geometry like fractals
- used by almost all 4k intros since ~2009
- see shadertoy.com for lots of examples

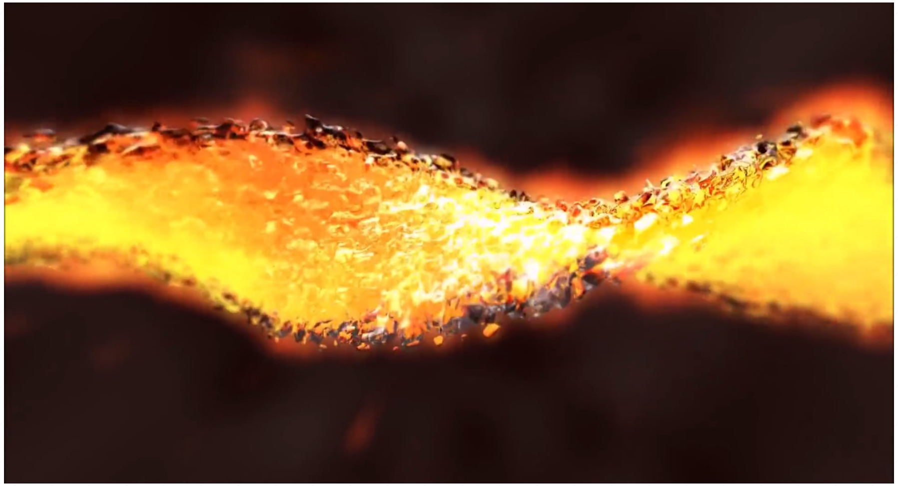
Example Shader

```
float bounce;
float sdBox(vec3 p,vec3 b) {
  vec3 d=abs(p)-b;
  return min(max(d.x,max(d.y,d.z)),0.)+length(max(d,0.));
}
void pR(inout vec2 p,float a) {
            p=cos(a)*p+sin(a)*vec2(p.y,-p.x);
}
float noise(vec3 p) {
            vec3 ip=floor(p);
    p-=ip;
    vec3 s=vec3(7,157,113);
    vec4 h=vec4(0.,s.yz,s.y+s.z)+dot(ip,s);
    p=p*p*(3.-2.*p);
    h=mix(fract(sin(h)*43758.5),fract(sin(h+s.x)*43758.5),p.x);
    h.xy=mix(h.xz,h.yw,p.y);
    return mix(h.x,h.y,p.z);
}
float map(vec3 p) {
            p.z-=1.0;
    p*=0.9;
    pR(p.yz,bounce*1.+0.4*p.x);
    return sdBox(p+vec3(0,sin(1.6*time),0),vec3(20.0, 0.05, 1.2))-.4*noise(8.*p+3.*bounce);
}
vec3 calcNormal(vec3 pos) {
    float eps=0.0001;
            float d=map(pos);
            return normalize(vec3(map(pos+vec3(eps,0,0))-d,
                   map(pos+vec3(0,eps,0))-d,map(pos+vec3(0,0,eps))-d));
}
float castRayx(vec3 ro,vec3 rd) {
    float function_sign=(map(ro)<0.)?-1.:1.;
    float precis=.0001;
    float h=precis*2.;
    float t=0.;
            for(int i=0;i<120;i++) {
        if(abs(h)<precis||t>12.)break;
                        h=function_sign*map(ro+rd*t);
        t+=h;
            }
    return t;
}
float refr(vec3 pos,vec3 lig,vec3 dir,vec3 nor,float angle,out float t2, out vec3 nor2) {
    float h=0.;
    t2=2.;
            vec3 dir2=refract(dir,nor,angle);
            for(int i=0;i<50;i++) {
                        if(abs(h)>3.) break;
                        h=map(pos+dir2*t2);
                        t2-=h;
            }
    nor2=calcNormal(pos+dir2*t2);
    return(.5*clamp(dot(-lig,nor2),0.,1.)+pow(max(dot(reflect(dir2,nor2),lig),0.),8.));
}
float softshadow(vec3 ro,vec3 rd) {
    float sh=1.;
    float t=.02;
    float h=.0;
    for(int i=0;i<22;i++) {
        if(t>20.)continue;
        h=map(ro+rd*t);
        sh=min(sh,4.*h/t);
        t+=h;
    }
    return sh;
}
void mainImage(out vec4 fragColor,in vec2 fragCoord) {
    bounce=abs(fract(0.05*time)-.5)*20.;
            vec2 uv=gl_FragCoord.xy/res.xy;
    vec2 p=uv*2.-1.;
    float wobble=(fract(.1*(time-1.))>=0.9)?fract(-time)*0.1*sin(30.*time):0.;
    vec3 dir = normalize(vec3(2.*gl_FragCoord.xy -res.xy, res.y));
    vec3 org = vec3(0,2.*wobble,-3.);
    vec3 color = vec3(0.);
    vec3 color2 =vec3(0.);
    float t=castRayx(org,dir);
            vec3 pos=org+dir*t;
            vec3 nor=calcNormal(pos);
    vec3 lig=normalize(vec3(.2,6.,.5));
    float depth=clamp((1.-0.09*t),0.,1.);
    vec3 pos2 = vec3(0.);
    vec3 nor2 = vec3(0.);
    if(t<12.0) {
            color2 = vec3(max(dot(lig,nor),0.)  +
                     pow(max(dot(reflect(dir,nor),lig),0.),16.));
            color2 *=clamp(softshadow(pos,lig),0.,1.);
            float t2;
                        color2.rgb +=refr(pos,lig,dir,nor,0.9, t2, nor2)*depth;
        color2-=clamp(.1*t2,0.,1.);
            }
    float tmp = 0.;
    float T = 1.;
    float intensity = 0.1*-sin(.209*time+1.)+0.05;
            for(int i=0; i<128; i++) {
        float density = 0.;
        float nebula = noise(org+bounce);
        density=intensity-map(org+.5*nor2)*nebula;
                        if(density>0.) {
                                    tmp = density / 128.;
            T *= 1. -tmp * 100.;
                                    if( T <= 0.) break;
                        }
                        org += dir*0.078;
    }
            vec3 basecol=vec3(1./1.  ,  1./4.  , 1./16.);
    T=clamp(T,0.,1.5);
    color += basecol* exp(4.*(0.5-T) - 0.8);
    color2*=depth;
    color2+= (1.-depth)*noise(6.*dir+0.3*time)*.1;
    fragColor = vec4(vec3(1.*color+0.8*color2)*1.3,
                     abs(0.67-depth)*2.+4.*wobble);
}
```

one scene from
"Rhodium" by Alcatraz
(1st @ Deadline 2016 PC 4k)

~ 3k uncompressed
(without comments, but with whitespace)

shadertoy.com/view/llK3Dy



Smaller Code

- CPU code is surprisingly large!
- for typical demos, a large part of this is *generic* libraries
  - e.g. music library that plays 20 formats with 4 different APIs
- for size-optimized code, *leave out everything you can!*
  - be sloppy: don't free memory, no sanity checks, no exceptions … YOLO!
- most importantly: *don't use the standard C/C++ library!*
  - 70k+ when linked statically, or a ~2 MiB DLL dependency
  - you don't need printf(), not even malloc(), and certainly not the STL
  - use the plain Win32 API where possible (malloc → HeapAlloc)
  - if a C library is absolutely required, use msvcrt.dll *(see the Crinkler manual for details)*
  - or write in assembly language (common for 4k intros)

Compression

- using all the techniques described so far,
  a typical 64k intro is still ~300k, a typical 4k intro like ~20k
- obvious solution: use an *executable compressor*
  - compresses the existing code and generates and EXE
    that decompresses everything on startup and runs it
- demoscene standard for 64k: kkrunchy
- demoscene standard for 4k: Crinkler
- different trade-offs: 4k can use extremely slow decompressor
- unfortunately, antivirus software flags everything compressed with these as totally evil, but that's another story …

Helping the Compressor

- ryg^Farbrausch: "Working With Compression" (2006)
- compression can be made more efficient when the uncompressed data contains as many repeating patterns as possible
- pre-process the data to better suit the compressor
  - *quantization:* store values with less precision where acceptable
  - *run-length encoding:* encode repeated values as "N times X" code
  - *delta coding:* only encode the difference to the previous value
  - *reordering:* group similar data together
    - music data, naive: sequence of events in <timestamp, channel, note> format
    - optimized: separate delta-coded timestamp and note data for each channel
  - put each type of data into its own section

Shader Minification

- remove stuff from shaders which is ignored by the compiler anyway
  - whitespace, comments, #ifdefs, { braces around single statements }, ...
  - rename local variables and functions to single letters
  - tedious to do by hand, but there's Ctrl+Alt+Test's Shader Minifier tool

```
for (CurStep = 1.0;
     ii >= 0.0 && t < 999.0 && CurStep > t * 0.000001;
     t += CurStep, RayStep = rayDir * t,  --ii) {
      CurStep = f(p+RayStep);
}
if (ii <= 1) {
      ii = 0.0;
      t = 999.3;
      RayStep = rayDir * t;
      break;
}
ii = smoothstep(44.0, 1.0, ii);
vec3 NextPos = p + RayStep;
p = NextPos;  // Start point and direction for reflected ray
CurNormal = vec2(0.04, 0.0);
vec3 n = vec3(
      fN(p + CurNormal.xyy) - fN(p — CurNormal.xyy),
      fN(p + CurNormal.yxy) - fN(p — CurNormal.yxy),
      fN(p + CurNormal.yyx) - fN(p - CurNormal.yyx));
```

```
for(t=1.;y>=0.&&t<999.&&t>t*1e-06;t+=t,v
=s*t,--y)t=f(e+v);if(y<=1){y=0.;t=999.3;
v=s*t;break;}y=smoothstep(44.,1.,y);vec3
m=e+v;e=m;i=vec2(.04,0.);vec3 x=vec3(f(e
+i.xyy)-f(e-i.xyy),f(e+i.yxy)-f(e-i.yxy)
,f(e+i.yyx)-f(e-i.yyx));
```

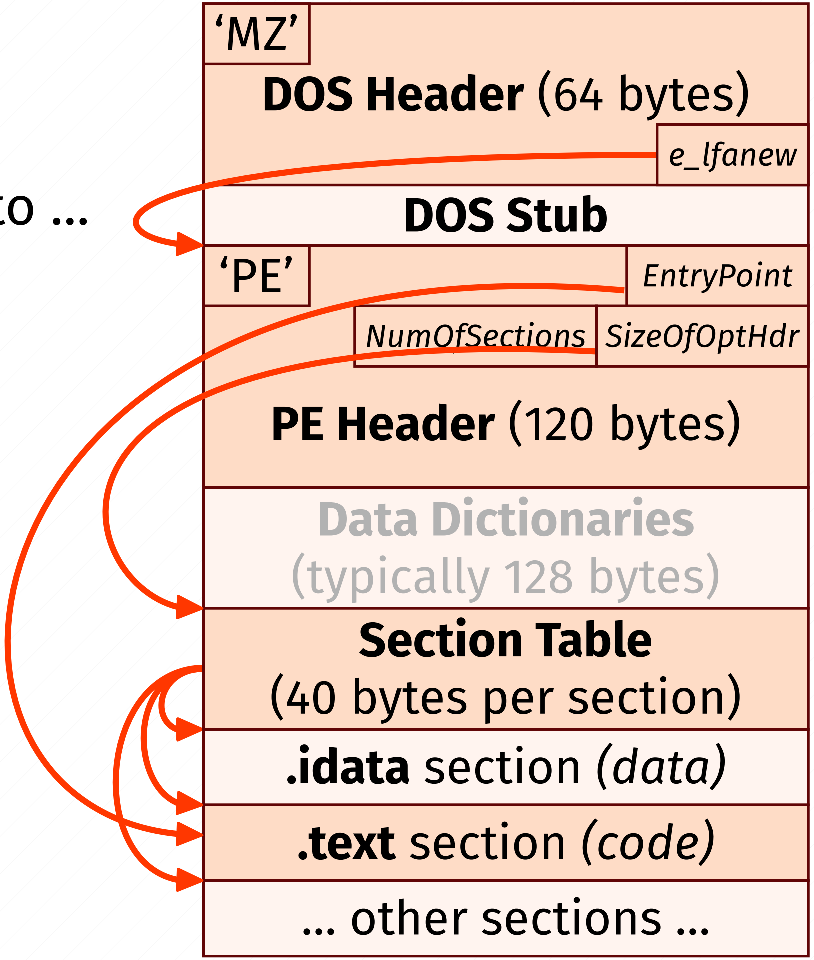
excerpt from the shader of BluFlame's 4k intro "Detached"

The Compressor’s Dirty Tricks

- kkrunchy and Crinkler don’t just compress the code, they also perform quite a few tricks to make unusually small EXEs
- kkrunchy: reordering and pre-processing for x86 code, some minor EXE header abuse, but mostly harmless™
  - transforms relative jumps into absolute jumps → more repetition!
- Crinkler uses every conceivable trick to make executables as small as possible
  - not always safe: older Crinkler-packed EXEs don’t always run on newer Windows versions
  - lots of extra-unsafe options to activate when space is getting *really* tight
- … but what do they actually *do*?

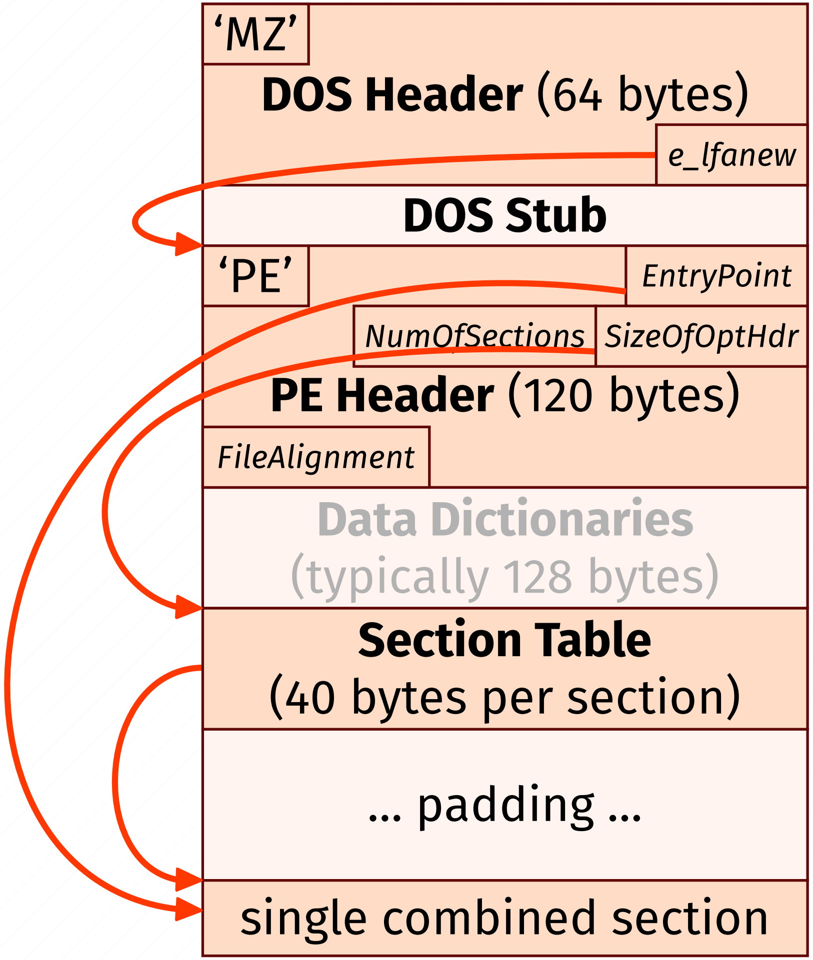
EXE File Structure

- every Windows EXE is also a DOS EXE
  - "DOS stub" prints message when run in DOS
  - a field at the end of the DOS header points to …
- PE header = "real" Windows EXE header
  - "Portable Executable"
- all actual code and data stored in **sections**
  - typically separate sections for code, initialized data, read-only data, etc.
  - but it's OK to just put everything into a single section



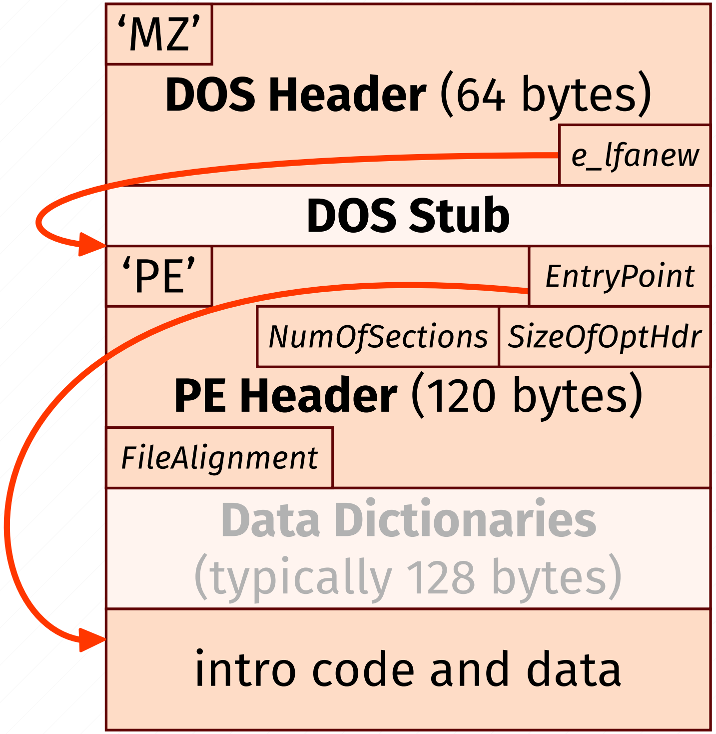
The Section Alignment Problem

- all sections must be *aligned* to at least 512 bytes
  - i.e. must start on a 512-byte boundary and be a multiple of 512 bytes in size
  - alignment specified in a field in the header ("FileAlignment")
- even a simple EXE with only header and a single section would need padding after the header!



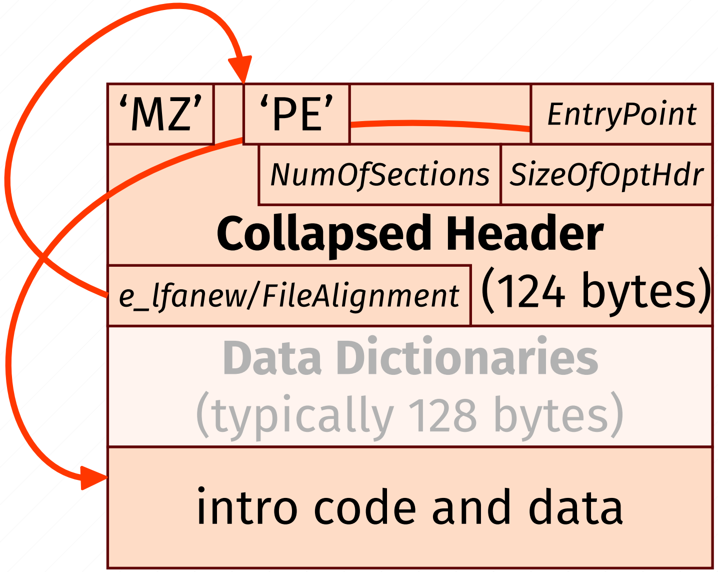
Sectionless Executables

- solution: don't use sections at all!
  - set NumberOfSections = 0
  - SizeOfOptHdr field (offset of section table) can be set to anything
    - ... in theory. In practice, it should be set to 8 to work around a bug(?) in Windows 7.
- also enables "low alignment mode"
  - FileAlignment can be as low as 1
  - can get rid of almost all padding!



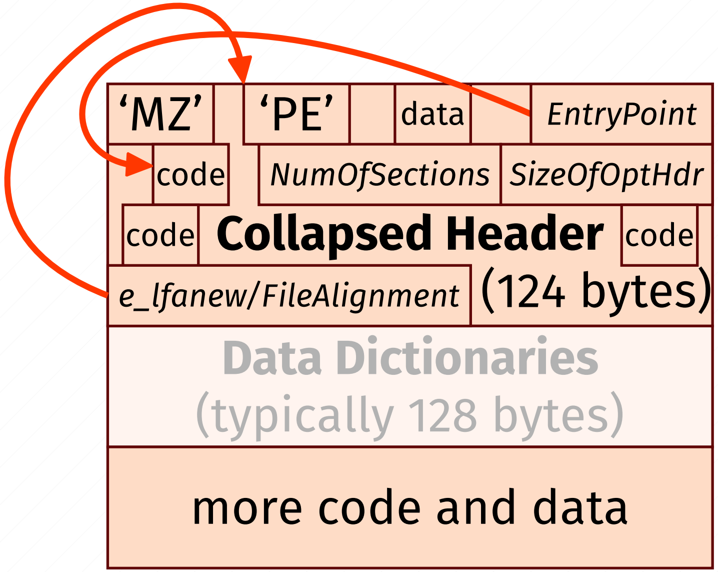
Collapsing Headers

- Windows mostly ignores the DOS header and stub
  - 'MZ' signature and PE offset are required, everything else is ignored
- trivial to remove the DOS stub
- possible to "collapse" the DOS header by moving the PE header *inside* it
  - e_lfanew (PE header offset) will then alias to some other field in the PE header
  - best solution: e_lfanew = FileAlignment = 4
    - only 2 unused bytes between DOS and PE header



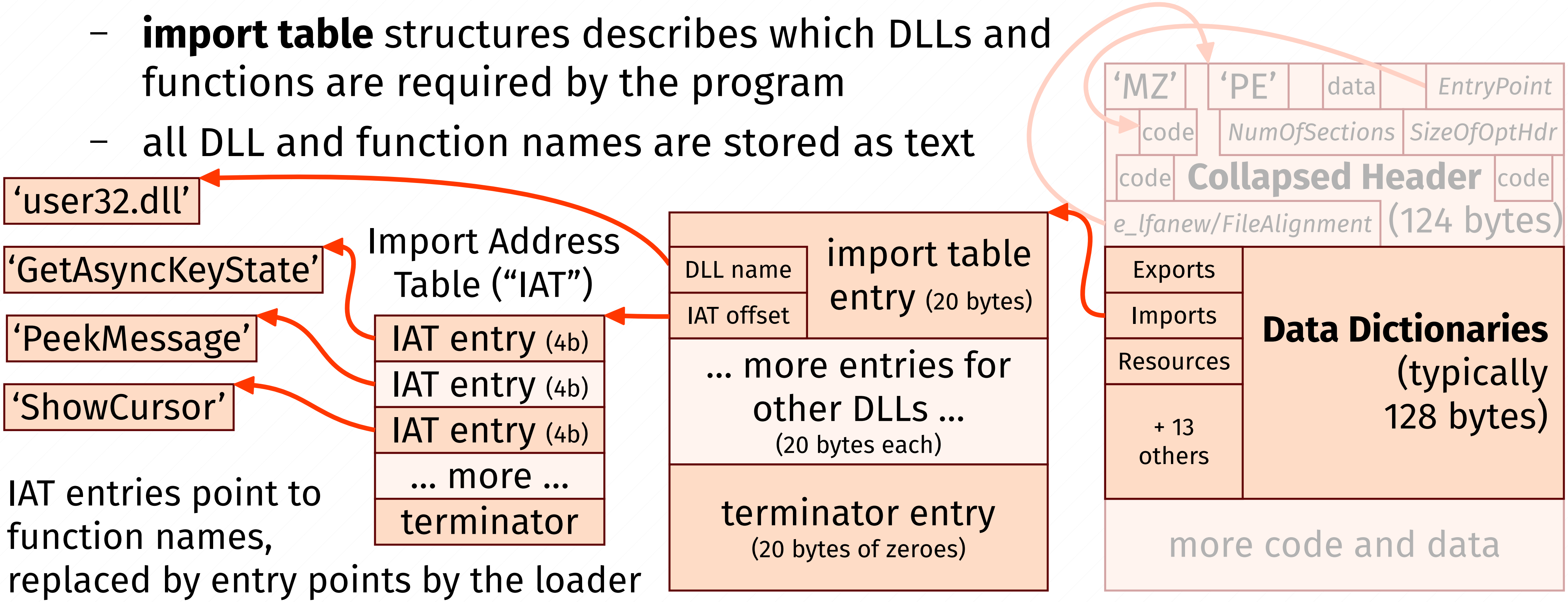
Ignored Fields

- PE header also contains lots of fields that are ignored by the loader
  - TimeDateStamp, LinkerVersion, SymbolTable, SizeOfData, BaseOfCode, BaseOfData, OperatingSystemVersion, ...
- possible to put useful code and data *directly into the header!*
  - only short snippets, but enough to be useful
  - entry point can even be inside the header



DLL Imports

- all useful programs require functions from other DLLs
  - **data dictionaries** contain offsets and sizes of various tables
  - **import table** structures describes which DLLs and functions are required by the program
  - all DLL and function names are stored as text



IAT entries point to function names, replaced by entry points by the loader

DLL Import Example

required DLL imports of a typical shader-based OpenGL 4k intro:

kernel32.dll	**user32.dll**	**gdi32.dll**	**winmm.dll**	**opengl32.dll**
ExitProcess	CreateWindowExA	SwapBuffers	waveOutOpen	wglCreateContext
CreateThread	GetDC	ChoosePixelFormat	waveOutPrepareHeader	glRects
	ShowCursor	SetPixelFormat	waveOutWrite	wglGetProcAddress
	PeekMessage		waveOutGetPosition	+ glCreateShaderProgramEXT + glUseProgram + glUniform4f *(but these must be loaded at runtime with wglGetProcAddress)*
	GetAsyncKeyState			
	ChangeDisplaySettingsA			

18 functions in 5 DLLs:
120 bytes of import tables
92 bytes of IATs
57 bytes of DLL name strings
281 bytes of function name strings
550 bytes total
… and these are part of the EXE header structures, can't be compressed!

Custom Importers

- the normal EXE import table structures are too large for intros
  - also, it doesn't even work for sectionless EXEs on Windows 8+!
- it's possible to do all this importing manually in code
  - when a DLL's base address in memory is known,
    we can parse the export table and look up the desired functions
  - LoadLibrary (from kernel32.dll) loads a DLL and returns its base address
  - kernel32.dll is loaded into *every* process,
    and its base address can be detected:

```
mov ebx, [fs:0x30]     ; get PEB pointer from TEB
mov ebx, [ebx+0x0C]    ; get PEB_LDR_DATA pointer from PEB
mov ebx, [ebx+0x14]    ; go to first LDR_DATA_TABLE_ENTRY
mov ebx, [ebx]         ; go to ntdll.dll's LDR_DATA_TABLE_ENTRY
mov ebx, [ebx]         ; go to kernel32.dll's LDR_DATA_TABLE_ENTRY
mov ebx, [ebx+0x10]    ; et voilà, kernel32.dll's base address!
```

- custom importer and its data
  can be part of the compressed code
- can also get rid of the Data Dictionaries in the PE header

Import By Hash

- the function names are still quite large, even when compressed
- only used to search for functions in some DLL's export table
- but if there's already a custom importer …
  - no strict need to match function names with normal string comparison
  - can do *anything* to uniquely identify each function
- don't store function names themselves, but a *hash* of the name
  - nothing fancy, just enough to tell function names in Windows DLLs apart
  - e.g. 32-bit xor-and-rotate hash:

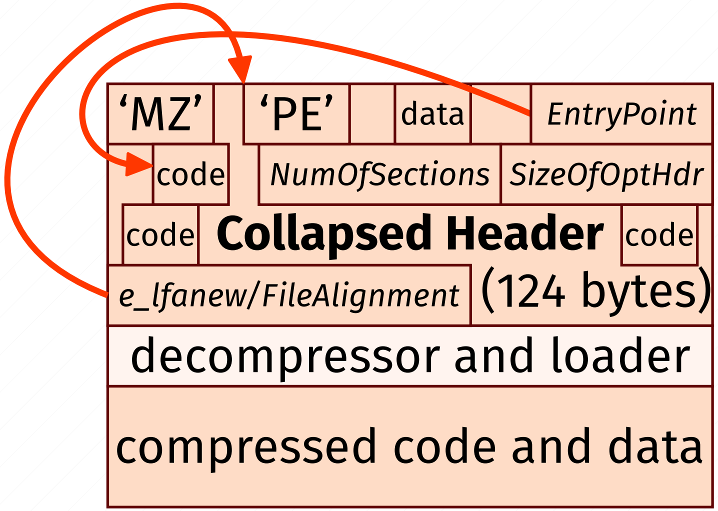
```
foreach (char c in functionName):
    hash = hash ^ c
    hash = (hash << 7) | (hash >> 25)
```

- from 14 bytes* per function down to 4!

* *(on average, uncompressed)*

Minimal EXE

- combining all the tricks, only the 124-byte header is left
  - plus decompressor and loader, around 200-300 bytes
  - 3,5k of (compressed) space to fill with awesome stuff!



So that's it!

- This concludes our overview of intro tricks and techniques.
- Thanks for your attention!
- Any questions?
- Get the slides at: https://keyj.emphy.de/intro-tricks/