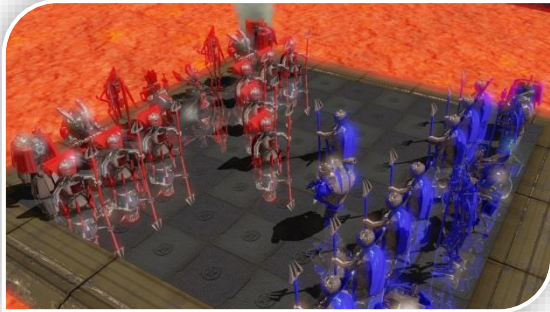


# Praktikum: Echtzeit Computergrafik



tum.3D  
computer graphics & visualization

- This week: Explosions!



## Task overview



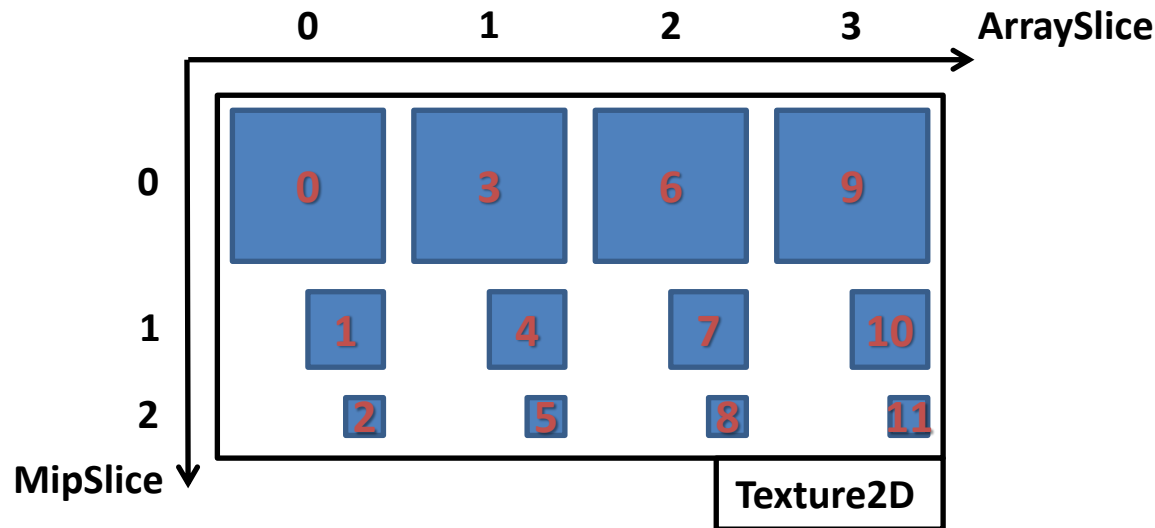
1. Extend your sprite renderer to support animated sprites
2. Spawn and update explosions
3. Create an additional particle system

- Sequence of keyframes instead of a single, static image



- Internally: 2D texture array instead of 2D texture
  - Texture2DArray != Array of Texture2D
  - Extend sprite vertices by parameters  $t, \alpha \in [0;1]$ 
    - $t$  indicates animation progress of the sprite
    - $\alpha$  is a scalar opacity factor for fading sprites in and out

- Each texture resource in D3D11 can consist of multiple subresources



- The **subresource index** can be calculated with `D3D11CalcSubresource()`
- Subresources can be initialized individually directly in `ID3D11Device::CreateTexture2D()`

```
HRESULT ID3D11Device::CreateTexture2D(  
    const D3D11_TEXTURE2D_DESC *pDesc,           //description of texture  
    const D3D11_SUBRESOURCE_DATA *pInitialData, //pointer to an array of subresources  
                                                //with pDesc->MipLevels*pDesc->ArraySize elements  
    ID3D11Texture2D **ppTexture2D               //output  
);
```

- pDesc->ArraySize: Size of array
- D3D11\_SUBRESOURCE\_DATA
  - Basically stores a pointer to subresource bytedata for initialization

```
typedef struct D3D11_SUBRESOURCE_DATA {  
    const void *pSysMem; //pointer to initialization data  
    UINT SysMemPitch;     //bytesize of a line in a 2D/3D texture  
    UINT SysMemSlicePitch; //bytesize of a xy-slice in a 3D texture  
} D3D11_SUBRESOURCE_DATA;
```

- Description for shader resource view needs to be set manually
  - ID3D11Device::CreateShaderResourceView( ..., const D3D11\_SHADER\_RESOURCE\_VIEW\_DESC \*pDesc, ...)

C++

```
typedef struct D3D11_SHADER_RESOURCE_VIEW_DESC {  
    DXGI_FORMAT          Format;  
    D3D11_SRV_DIMENSION ViewDimension;  
    union {  
        D3D11_BUFFER_SRV      Buffer;  
        D3D11_TEX1D_SRV       Texture1D;  
        D3D11_TEX1D_ARRAY_SRV  Texture1DArray;  
        D3D11_TEX2D_SRV        Texture2D;  
        D3D11_TEX2D_ARRAY_SRV  Texture2DArray;  
        D3D11_TEX2DMS_SRV      Texture2DMS;  
        D3D11_TEX2DMS_ARRAY_SRV Texture2DMSArray;  
        D3D11_TEX3D_SRV         Texture3D;  
        D3D11_TEXCUBE_SRV        TextureCube;  
        D3D11_TEXCUBE_ARRAY_SRV TextureCubeArray;  
        D3D11_BUFFEREX_SRV      BufferEx;  
    };  
} D3D11_SHADER_RESOURCE_VIEW_DESC;
```



- Format = „texture format“ (DXGI\_FORMAT\_\*, from texture description)
- ViewDimension = D3D11\_SRV\_DIMENSION\_TEXTURE2DARRAY
- Texture2DArray:
  - ArraySize = „size of array“ (from texture description)
  - MipLevels = „number of miplevels“ (from texture description)
  - FirstArraySlice = MostDetailedMip = 0

C++

```
typedef struct D3D11_SHADER_RESOURCE_VIEW_DESC {  
    DXGI_FORMAT      Format;  
    D3D11_SRV_DIMENSION ViewDimension;  
    union {  
        D3D11_BUFFER_SRV      Buffer;  
        D3D11_TEX1D_SRV      Texture1D;  
        D3D11_TEX1D_ARRAY_SRV Texture1DArray;  
        D3D11_TEX2D_SRV      Texture2D;  
        D3D11_TEX2D_ARRAY_SRV Texture2DArray;  
        D3D11_TEX2DMS_SRV     Texture2DMS;  
        D3D11_TEX2DMS_ARRAY_SRV Texture2DMSArray;  
        D3D11_TEX3D_SRV      Texture3D;  
        D3D11_TEXCUBE_SRV     TextureCube;  
        D3D11_TEXCUBE_ARRAY_SRV TextureCubeArray;  
        D3D11_BUFFEREX_SRV    BufferEx;  
    };  
} D3D11_SHADER_RESOURCE_VIEW_DESC;
```



- CreateDDSTextureFromFile() takes care of all this!
  - Loads texture arrays from single DDS
  - Creates resource and resource views
- DirectXTex: texAssemble command line tool
  - Generates 2D Arrays / Cubemaps from multiple images
  - Outputs a single DDS
  - Add the texAssemble project to your solution...
  - ... and call it in your ResourceGenerator

- TexAssemble requires explicit specification of all images
  - We'll provide you a batch file to convert complete folders
  - See assignment
- TexAssemble does not provide MipMap-Generation and compression
  - Call texconv on the resulting DDS!

- Texture object declaration

```
Texture2DArray g_SpriteTex;
```

HLSL

- Binding shader resource variable (as usual)

```
m_pEffect->GetVariableByName("g_SpriteTex")->AsShaderResource()  
->SetResource(...);
```

C++

- Sampling the array

```
//position.xy: texture coordinates  
//position.z : array index  
float3 position;  
g_SpriteTex.Sample(samAnisotropic, position);
```

HLSL

- Array index position.z: rounded to next integer (array index)
- Use GetDimensions() to get the texture size in the shader

```
float3 dims; //(x-width, y-height, arraysize)  
g_SpriteTex.GetDimensions(dims.x, dims.y, dims.z); //dims is written here!
```

HLSL

- How to handle multiple sprite texture indices?
  - Only a single draw call for all particles (sorting!)
  - Use an array with a reasonable upper bound

```
Texture2DArray g_SpriteTex[5];
```

- Binding resources, two alternatives:

```
m_pEffect->GetVariableByName("g_SpriteTex")->AsShaderResource()  
->SetResourceArray(...);
```

```
m_pEffect->GetVariableByName("g_SpriteTex")->GetElement(index)  
->AsShaderResource()->SetResource (...);
```

- HLSL does not support texture fetches with dynamic indices
  - `switch()` with all possible indices does the job...
  - Won't win a price for the most beautiful code, though

- Same procedure as for your projectiles
- Add each explosion to a global list
  - Store texture index, position and current animation state `t`
  - Each frame, advance the animation state of all explosions
  - Use `fElapsedTime` and the total duration
  - Don't forget to delete explosions!
- On render, add all your explosions to the list of `SpriteVertex`
  - Sort and render along with your projectiles

- Adding some fancyness: additional particles

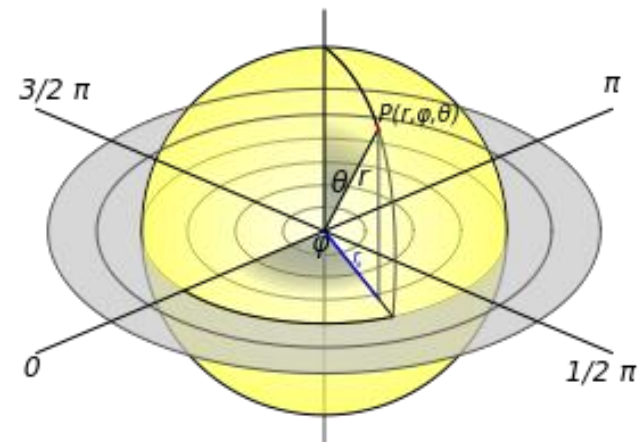


- Create a number of „explosion particles“ at the center
  - Again, like your projectiles!
  - Update (speed, position, animation?, alpha?), render
- Storage
  - Either use a new global container for this
  - ... or unify all properties (explosion, particles, projectiles)
  - Using seperate containers might be better for experimenting though
- Initialization
  - Random direction
  - Random speed, size & position?



- Some hints for random directions
  - See the slides for Assignment 2 for random number generators and distributions!
  - Randomizing  $x, y, z$  in  $[-1, 1]$  will result in a bad distribution
  - Alternative 1: **Rejection sampling**, discard if  $\text{length}((x, y, z)) > 1$
  - Alternative 2: **Spherical coordinates**, two random angles  $\theta$  in  $[0^\circ; 180^\circ]$  and  $\varphi$  in  $[0^\circ; 360^\circ]$ :

$$\begin{pmatrix} x \\ y \\ z \end{pmatrix} = \begin{pmatrix} \sin(\theta) \cdot \cos(\varphi) \\ \sin(\theta) \cdot \sin(\varphi) \\ \cos(\theta) \end{pmatrix}$$



# Questions?

(we're not done, don't run away yet!)

Maybe it's time to address  
your particle system  
performance.

At least a tiny bit.

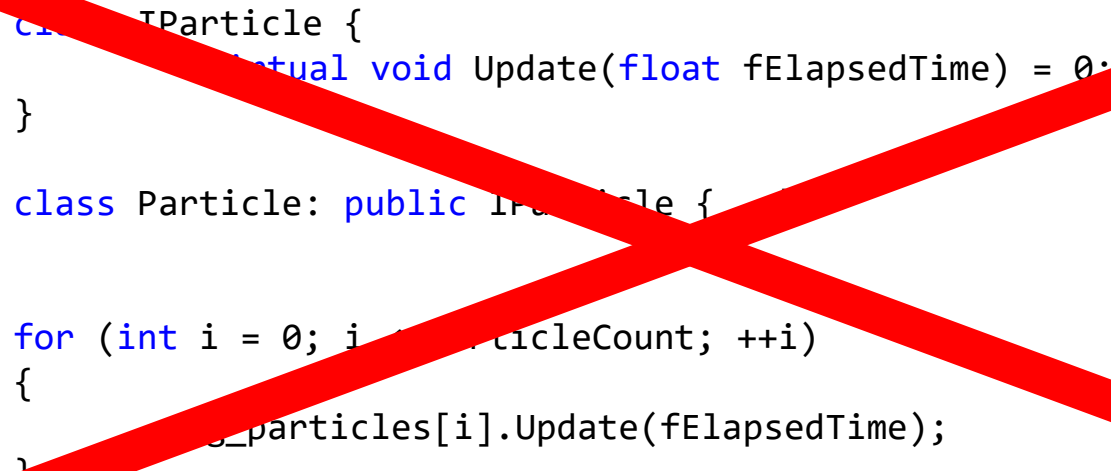
**Study at home!**



- Keep an eye on your function calls
  - Function calls aren't for free
  - Calling overridden functions is more expensive!
- Worst case scenario:

```
class IParticle {  
    virtual void Update(float fElapsedTime) = 0;  
}  
  
class Particle: public IParticle { ... }  
  
for (int i = 0; i < particleCount; ++i)  
{  
    g_particles[i].Update(fElapsedTime);  
}
```

- Keep an eye on your function calls
  - Function calls aren't for free
  - Calling overridden functions is more expensive!
  - Worst case scenario:



```
class IParticle {  
    virtual void Update(float fElapsedTime) = 0;  
}  
  
class Particle: public IParticle {  
  
    for (int i = 0; i < particleCount; ++i)  
    {  
        _particles[i].Update(fElapsedTime);  
    }  
}
```

- Keep an eye on your function calls
  - Instead:

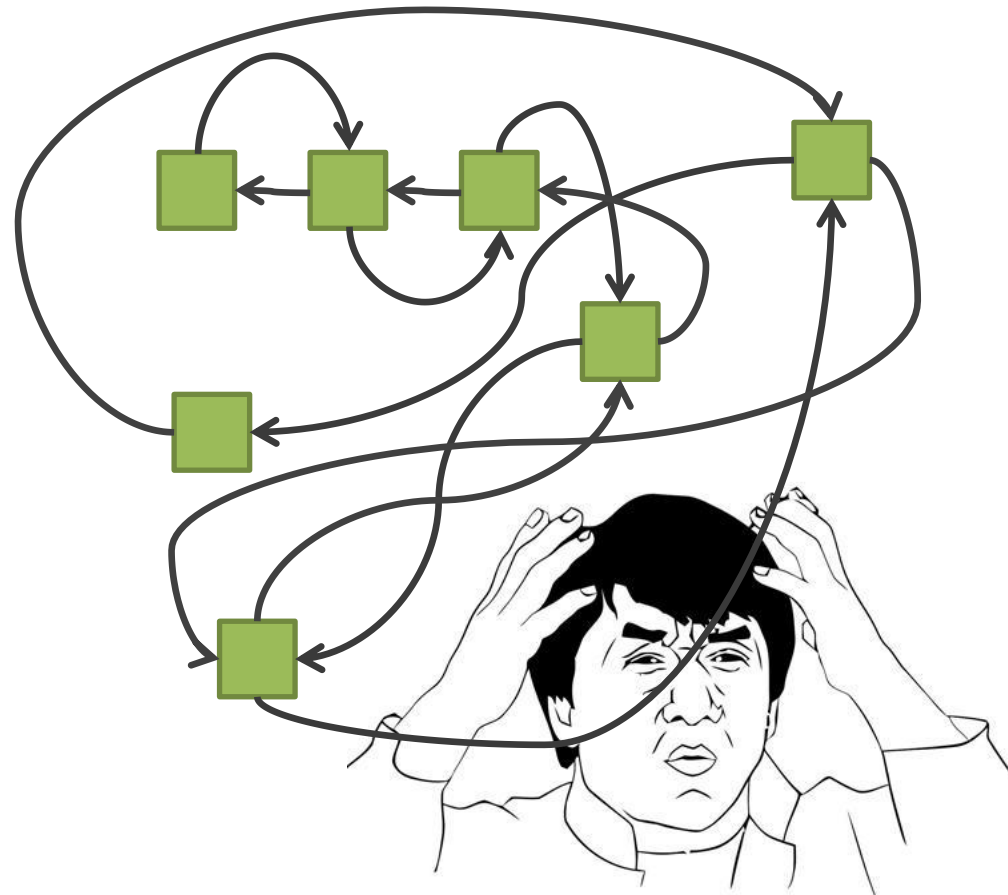
```
struct Particle {  
    XMFLOAT3 m_position  
    // Speed, etc  
}  
  
for (int i = 0; i < particleCount; ++i)  
{  
    g_particles[i].m_position = ...;  
}
```

## `std::vector` VS `std::list`

- Array of elements
- Consecutively stored in memory
- Single pointer to first element

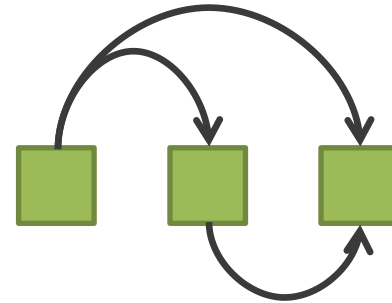


- Each elements points to next & prev
- Can be anywhere in memory





- `std::list`
  - Removal at a random position is fast
  - Iteration over all elements is „okay-ish“
  - No random access – always iterate over all elements
  - Adding elements at random positions is okay, but memory allocations / deallocations can kill you



- `std::vector`
  - Iteration over all elements is fast
  - Random access is fast
  - Appending elements is fast - most of the time!
    - All elements are copied when no more space is available
    - Amount of reserved memory is doubled -> minimizes reallocation operations
  - Removal of the last element is fast
  - Removal of intermediate elements (`erase()`) is deadly...

- `std::vector`
  - Iteration over all elements is fast
  - Random access is fast
  - Appending elements is fast - most of the time!
    - All elements are copied when no more space is available
    - Amount of reserved memory is doubled -> minimizes reallocation operations
  - Removal of the last element is fast
  - Removal of intermediate elements (`erase()`) is deadly...



- Avoid removal of intermediate elements in `std::vector`
  - For „bulk removal“: instead of removing an element, „cut and paste“ the last one

```
int count = vec.size();
for (int i = 0; i < count; ++i) {

    bool remove = ...;
    if (remove) {
        --count;
        vec[i] = vec[count];
        --i;          // Check the same index again
    }

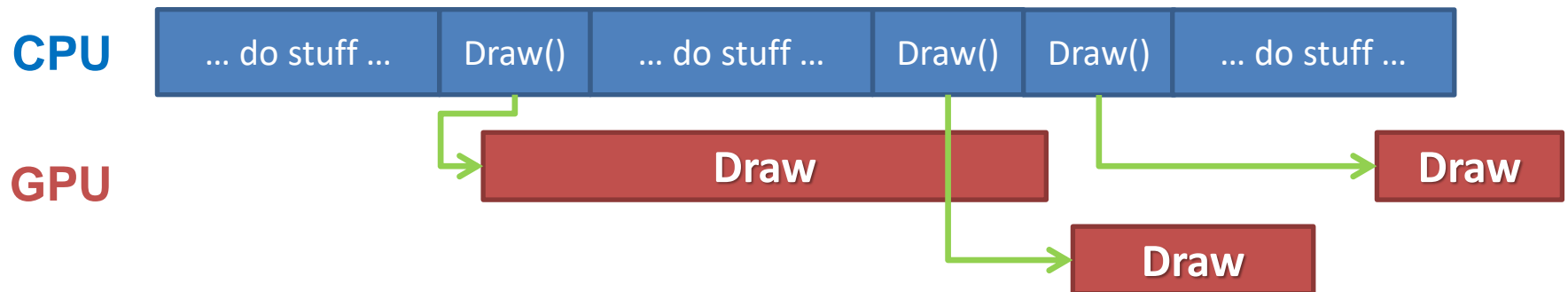
}

vec.resize(count);
```


- `std::map` and `std::unordered_map`
  - Use those if you need a „dictionary“ style container
- `std::map`
  - (Binary) tree
  - Elements have an order
  - Access time:  $O(\log n)$
- `std::unordered_map`
  - Hash map
  - Elements have no order
  - Access time: „Usually“ constant,  $O(1)$

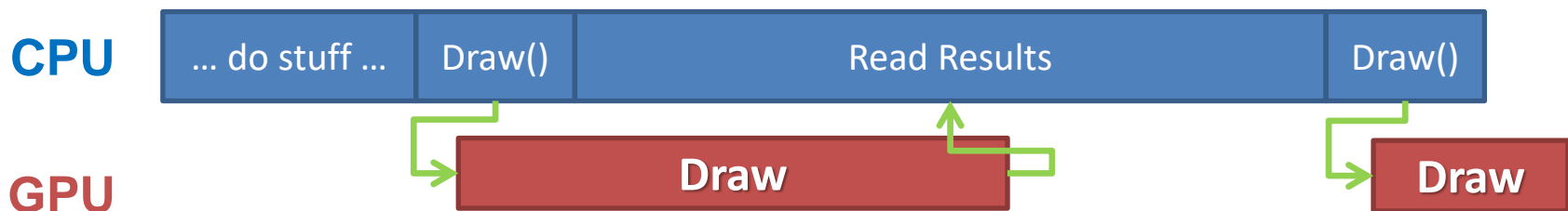
- `std::vector` is the best choice most of the time
  - Keep in mind how the data is organized
  - Avoid `erase()`
  - Reserved memory isn't released if you resize a vector
    - So appending elements is basically free at some point
    - Only if your vector isn't a local variable!
- Always iterate „front to back“ over your data
  - Access the data in the order it's stored in memory
  - Keyword „caching“: The next values might already be stored in the fast CPU caches


- Keep in mind that data is processed in **parallel** and **asynchronously** on the GPU
  - Parallelism: enough data must be available to process
  - Asynchronism: synchronization must be minimized





- Keep the GPU busy!
  - Reduce the number of draw calls
    - In fact: Increase the data per draw call to avoid GPU idle time
  - Avoid synchronization points
    - GPU output is CPU input 
      - Could you use the data from the previous frame?
      - Could you draw something else during the idle-time?
    - Sidenote: Of course this cannot be completely avoided...



- Keep the GPU busy!
  - Reduce the number of draw calls
    - In fact: Increase the data per draw call to avoid GPU idle time
  - Avoid synchronization points
    - GPU output is CPU input 
      - Could you use the data from the previous frame?
      - Could you draw something else during the idle-time?
    - Sidenote: Of course this cannot be completely avoided...
  - Avoid overdraw
    - Many large, transparent particles are expensive

- Avoid unnecessary calculations in the shader:  
Precalculate on the CPU
  - Combine transformation matrices

```
T3dVertexPSIn MeshVS(T3dVertexVSIn Input)
{
    T3dVertexPSIn Output = (T3dVertexPSIn) 0;
    Output.Pos = mul(float4(Input.Pos, 1.f), g_matObject);
    Output.Pos = mul(Output.Pos, g_matAnim);
    Output.Pos = mul(Output.Pos, g_matWorld);
    Output.Pos = mul(Output.Pos, g_matView);
    Output.Pos = mul(Output.Pos, g_matProjection);

    return Output;
}
```



```
T3dVertexPSIn MeshVS(T3dVertexVSIn Input)
{
    T3dVertexPSIn Output = (T3dVertexPSIn) 0;
    Output.Pos = mul(float4(Input.Pos, 1.f), g_WorldViewProjection);
    return Output;
}
```

- Precalculate per-frame constant factors in complex formulas

- Shaders are executed on SIMD Units
  - Single instruction, multiple data
  - Consequence: „Multiple data“ must be available to run the same instruction on...
- **Coherence** is important: Works best if all data is processed the same way
- Each group is as slow as its slowest member...
  - Avoid divergent branches
  - Avoid pixel shaders with highly varying local complexity

## Always check your performance without a debugger!

- Build in Release configuration
- Start with **Ctrl + F5**
- `std` containers are quite slow in Debug build

# Questions?