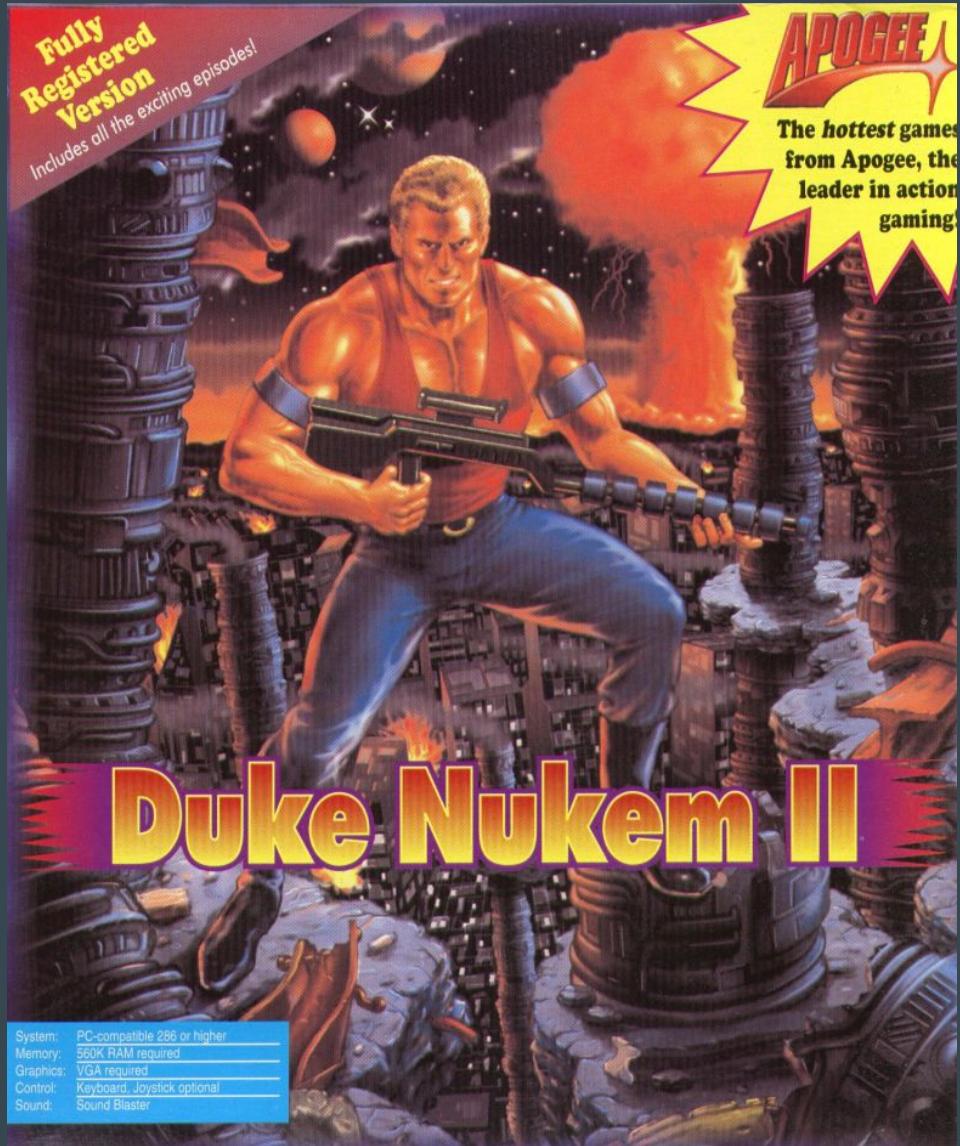


# Rigel Engine

Reimplementing an old DOS game in modern C++

The game: Duke Nukem II



## Duke Nukem II

Released: Dec. 1993

Free shareware version (1/4 of full content)

Paid full version (\$ 35)

System requirements:

- 286 CPU (386 recommended)
- 560 KB RAM
- MS-DOS 3.3
- VGA-compatible video card

Optionally supported:

- Soundblaster + AdLib sound cards
- Joysticks



## Duke Nukem II

Size on disk (executable + data):

- 3.4 MB (shareware)
- 5.7 MB (full version)

Audio/video output:

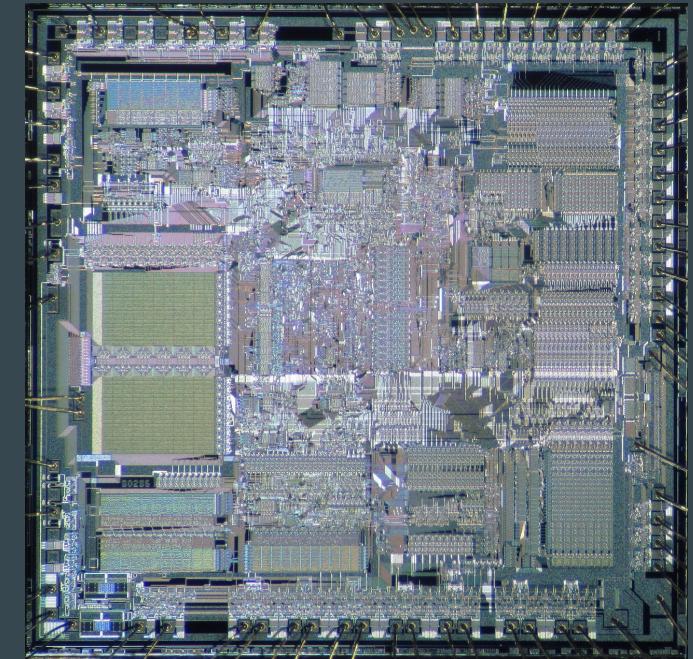
- 320x200 screen resolution (stretched to 4:3 by CRT)
- 16 color palette (256 for intro movies)
- 15-17 FPS
- Monoaural sound output
- No mixing (only 1 sound effect at a time)

# Hardware of the time

# CPUs

## 80286 (1982)

- Clock speed: 4 to 12.5 MHz, later 20 to 25 MHz
- At 12 MHz: ~2.66 MIPS
- No L1/L2 cache
- Only 16-bit registers
- No floating point (unless co-processor installed)



# CPUs

## 80386 (1985)

- Clock speed: 12 to 40 MHz
- At 33 MHz: ~9.9 MIPS
- Optional cache (out-of-chip)

## 80486 (1989)

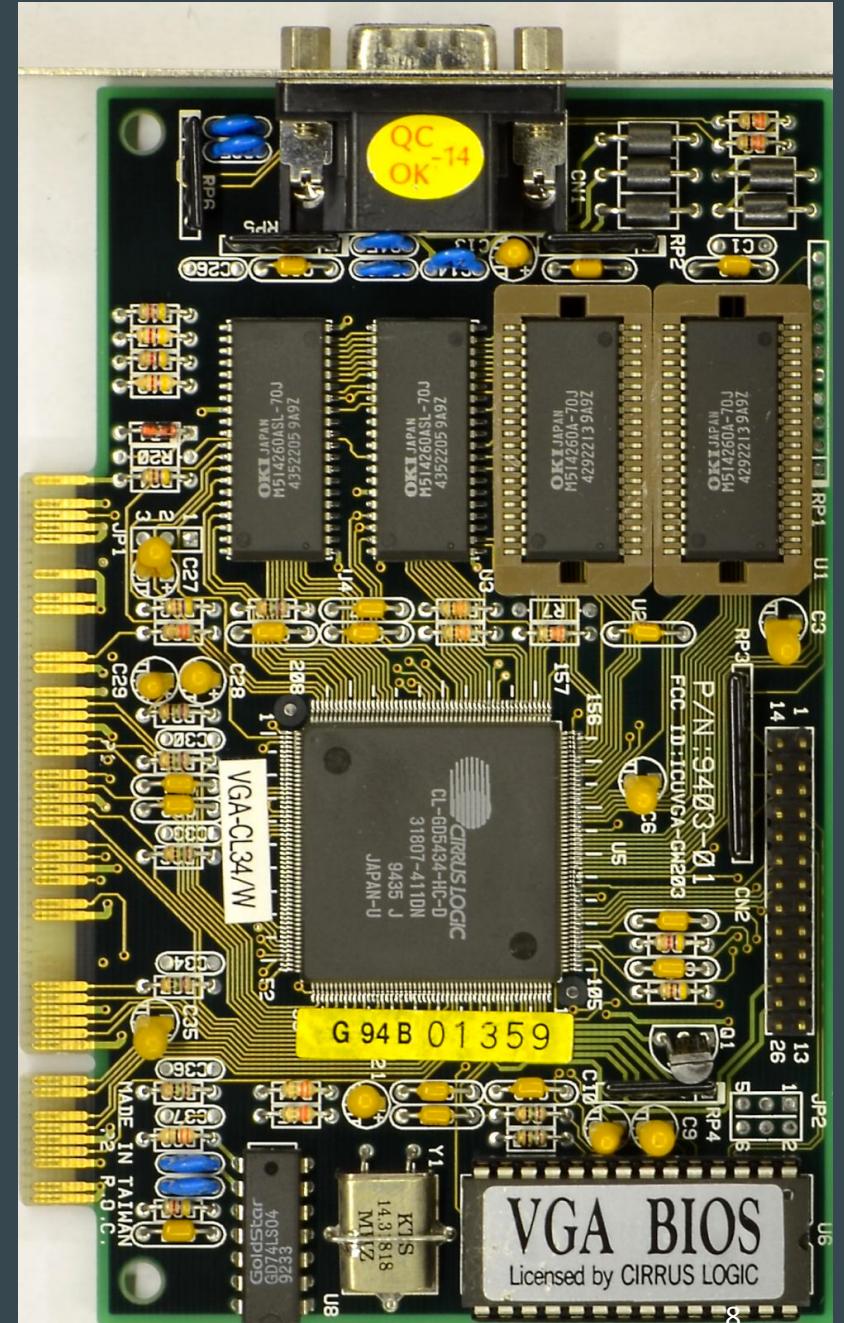
- On-chip cache, improved pipelining, integrated floating point unit
- Clock speed: 20 to 66 MHz, 100 MHz
- At 50 MHz: ~40 MIPS



# VGA Graphics

- First release: 1987
- (min.) 256 KB VRAM
- 16 or 256 colors
- Common resolutions:
  - 640x480
  - 640x350
  - 640x200
  - 320x200

[http://www.vgamuseum.info/images/palcal/cirrus/669\\_vga-cl34\\_w\\_cirrus\\_logic\\_cl-5434-hc-d\\_top\\_hq.jpg](http://www.vgamuseum.info/images/palcal/cirrus/669_vga-cl34_w_cirrus_logic_cl-5434-hc-d_top_hq.jpg)



# AdLib and Sound Blaster sound

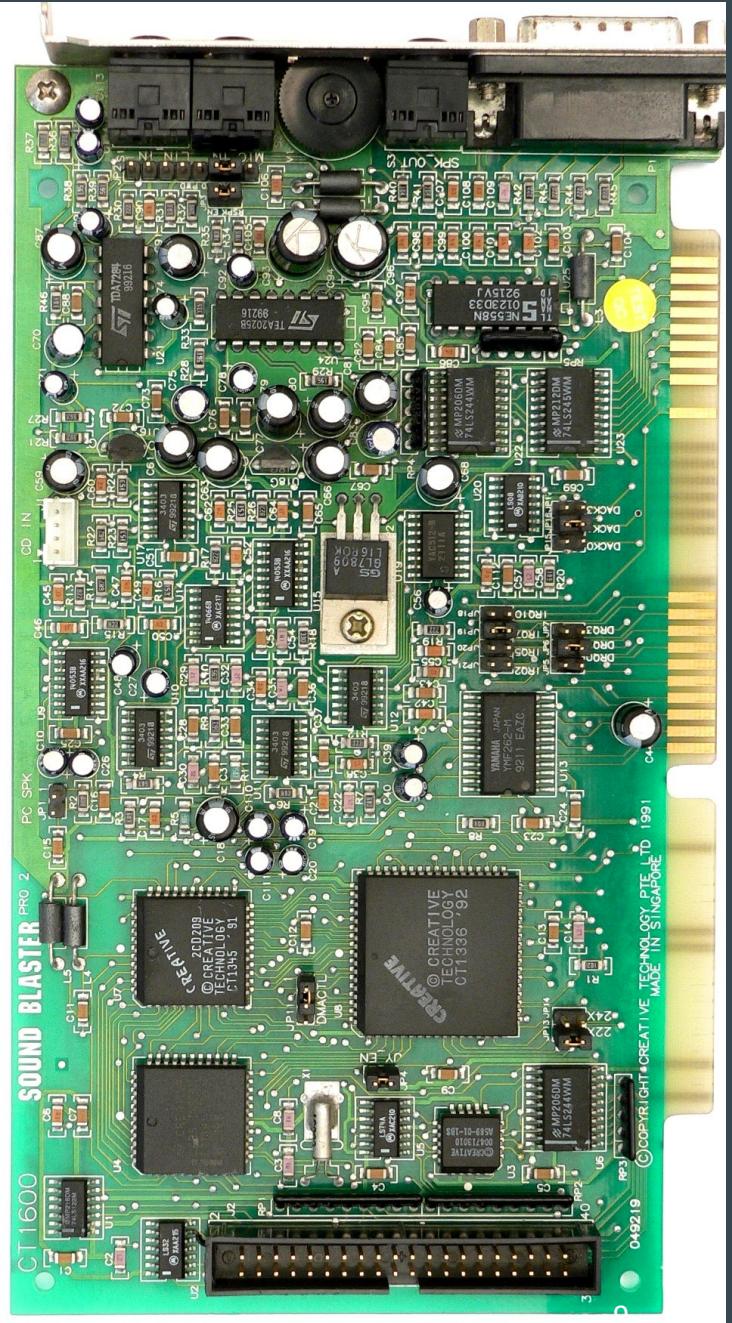
## AdLib (1987)

- Add-on card with FM synthesizer chip
- Intended for music, but can do some sounds

## Sound Blaster (1989), Sound Blaster Pro (1991)

- Full compatibility to AdLib
  - Includes the same synthesizer chip
- Additional support for digitized sound playback
  - 8-bit, 22.05 kHz (44.1 kHz on SB Pro)

[http://www.mediafire.com/convkey/44ba/4qwdgeqjuummj35zg.jpg?size\\_id=a](http://www.mediafire.com/convkey/44ba/4qwdgeqjuummj35zg.jpg?size_id=a)



# The project: Rigel Engine

# What is Rigel Engine?

- 2D game engine based on OpenGL and Entity-Component-Systems
- Code to read assets (data) from original game's files
  - Graphics, sounds, music
  - Maps/level data
- Implementation of original game logic/enemy behaviors etc.
  - Based on reverse-engineering

All written in modern C++

# What is it not?

- An emulator\*
  - Logic is completely re-implemented in native code
- Recreating the original source code/architecture
  - Aiming for equal behavior, but very different architecture under the hood
- Able to run on the original hardware
  - Being free of the (very tight) constraints is part of the fun

\* Emulation is used to play back music

# Advantages of a modern re-implementation

- No emulator necessary – fully native
  - More responsive controls
  - Shorter loading times
  - Proper gamepad/controller support
- Possibility for enhanced graphics and sound
- Software preservation – source code allows:
  - Porting to future platforms
  - Bug fixes
  - Enhanced modding support

# Similar projects

<https://davidgow.net/keen/omnispeak.html>

*Omnispeak is an open-source reimplementation of Commander Keen episodes 4, 5, and 6. It aims to be a pixel-perfect, bug-for-bug clone of the original games, and is compatible with savegames from the DOS version.*

<http://clonekeenplus.sourceforge.net/>

*Commander Genius is a software piece that interprets the Commander Keen Invasion of the Vorticons and Galaxy series. As fans and developers, we are, we try to implement new features, improve the gameplay, and give players an experience that feels like playing the original game but a bit more refreshing.*

# Quick project overview

- C++ 14 (migration to 17 planned)
- ~ 18,000 LOC (2.4k LOC tests)
- GPL v2
- CMake
- CI via Travis CI/AppVeyor

# Quick project overview

- OpenGL 3.0 (or ES 2.0)
- SDL (<http://www.libsdl.org/>)
- Boost (optional, variant, program options)
- EntityX (<https://github.com/alecthomas/entityx>)
- DBOPL AdLib emulator (from DosBox, <https://www.dosbox.com/>)

# Project evolution

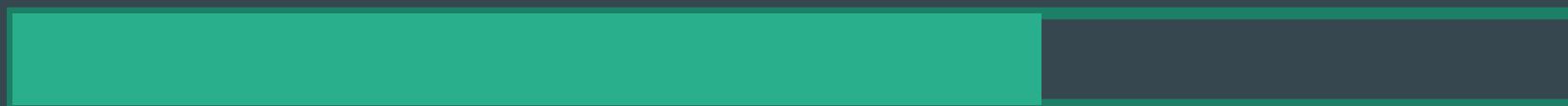
- First commit: 21 Aug 2016 (On GitHub: 22 Oct)
- 1007 commits (760 on GitHub)
- Total time worked on: 14-15 months

# Implementation progress (Shareware only)

14/24 enemies



20/29 game mechanics



6/15 non-ingame features



# Implemented enhancements

- Concurrent sound effects
- No loading screens
- More responsive controls
- Removed limitations
  - Max number of particles
  - Max number of explosions, debris and other effects

# Supported platforms

- Windows (MSVC 2015)
- Linux (Clang 3.9, GCC 5.4)
- Mac OS X (AppleClang 8.0.0)
- Raspberry Pi (GCC 5.4 for ARM)

# Live Demo!

Game Logic

6.6k LOC

Asset loader

2.5k LOC

Engine

2.2k LOC

UI

1.9k LOC

OpenGL renderer

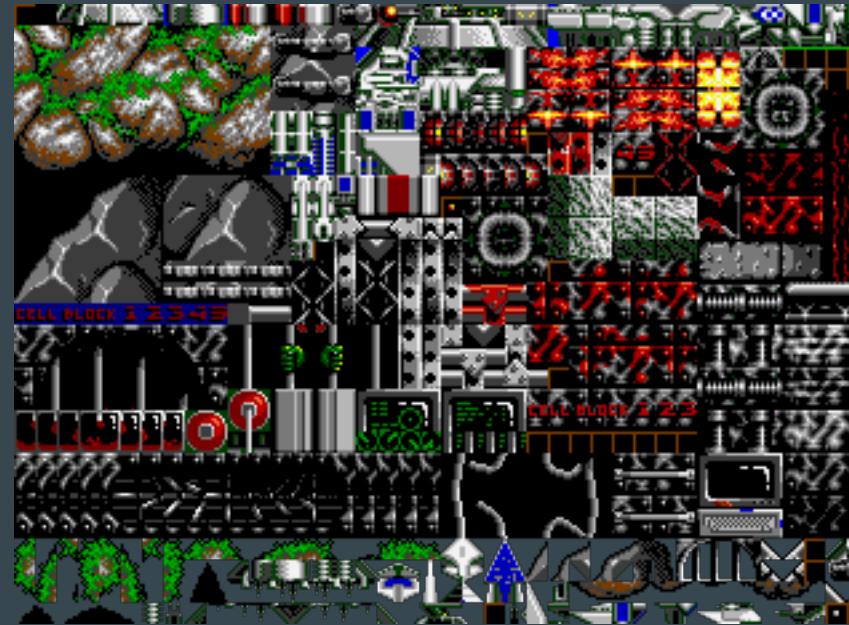
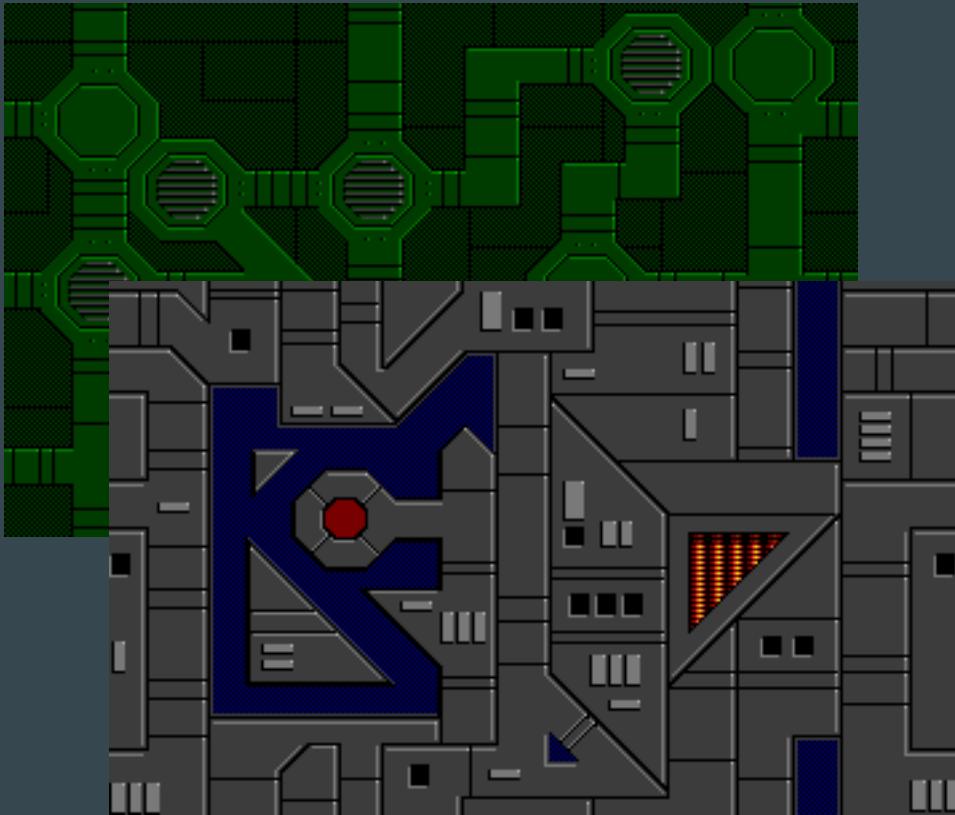
1.3k LOC

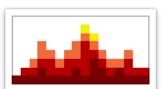
# Asset loading

# What's in the game data?

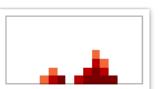
- Graphics
  - Sprites
  - Parallax backgrounds
  - Tile sets
  - Full-screen images
- Sounds
- Music
- Level data
- Menu descriptions

# Example: Backgrounds, tile set

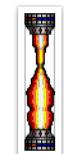




65\_9.png



65\_10.png



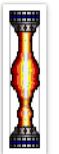
66\_0.png



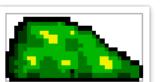
66\_1.png



66\_2.png



66\_3.png



67\_0.png



67\_1.png



67\_2.png



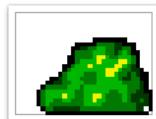
67\_3.png



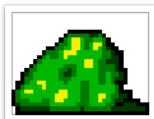
67\_4.png



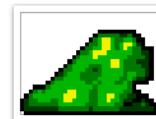
67\_5.png



67\_6.png



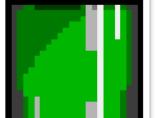
67\_7.png



67\_8.png



67\_9.png



68\_0.png



68\_1.png



68\_2.png



68\_3.png



68\_4.png



68\_5.png



68\_6.png

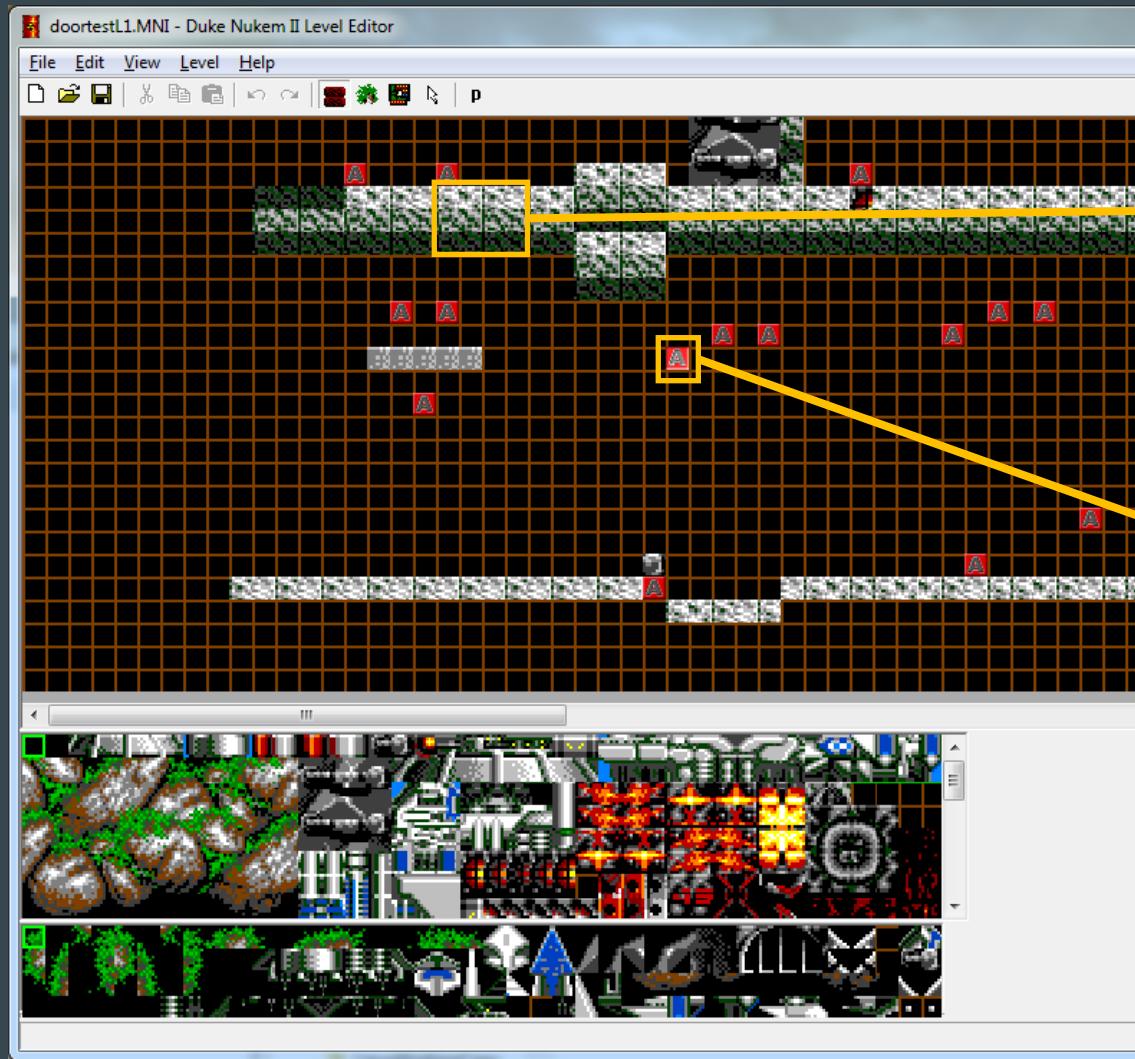


68\_7.png



68\_8.png

# Example: Level data



Tile indices

422, 423, 424, 425,  
462, 463, 464, 465,

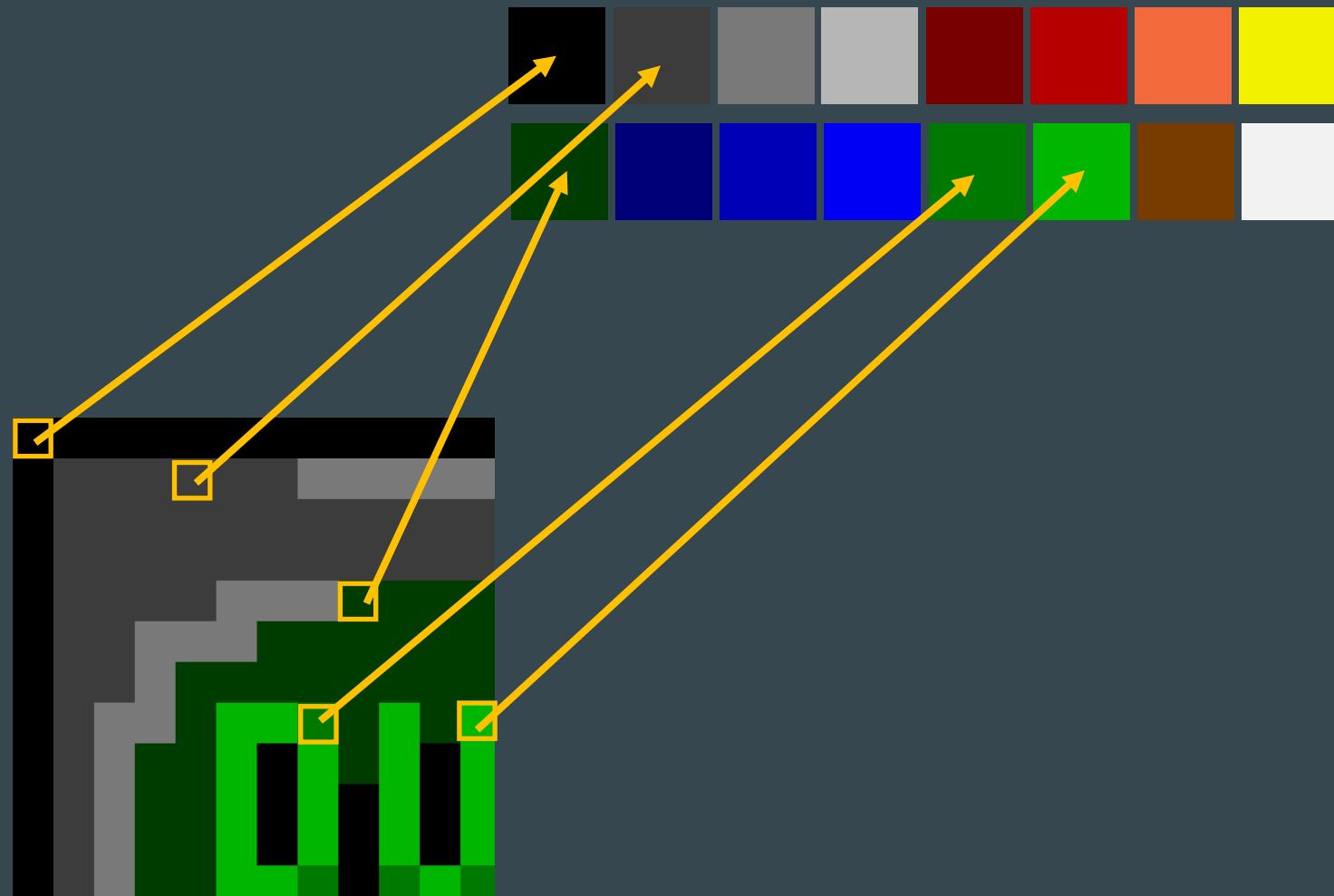
...

Type: 154

PosX: 35

PosY: 100

# Palette-based graphics

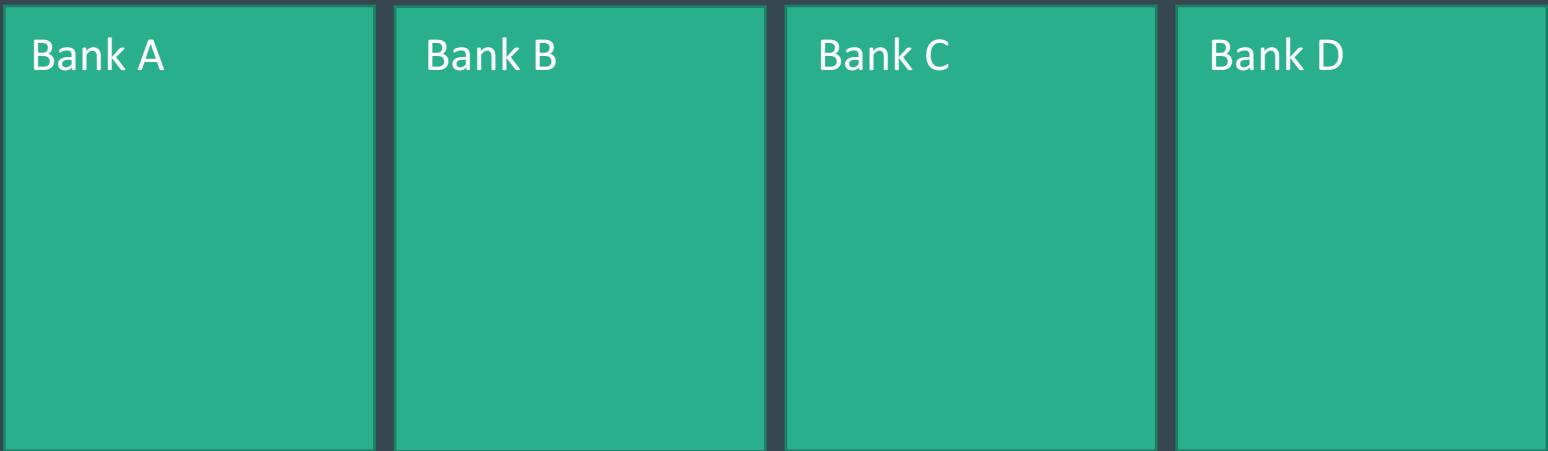


16 color palette  
→ need 4 bit per pixel  
→ 2 pixels per byte

# Planar memory layout



8, 13, 13, 12



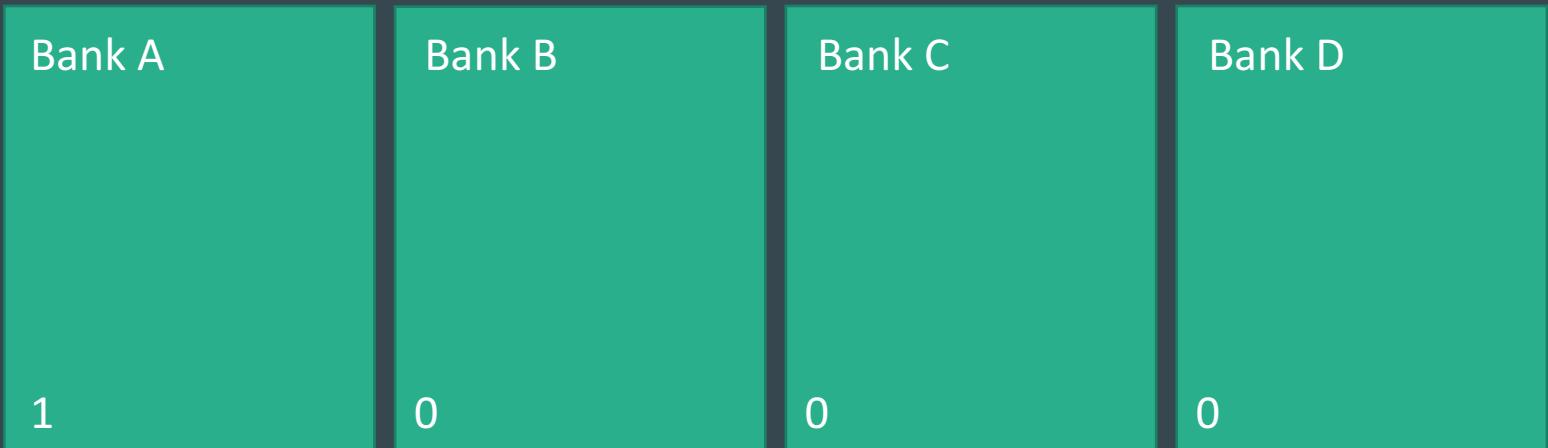
# Planar memory layout



8, 13, 13, 12



$8 \rightarrow 1000_2$



# Planar memory layout



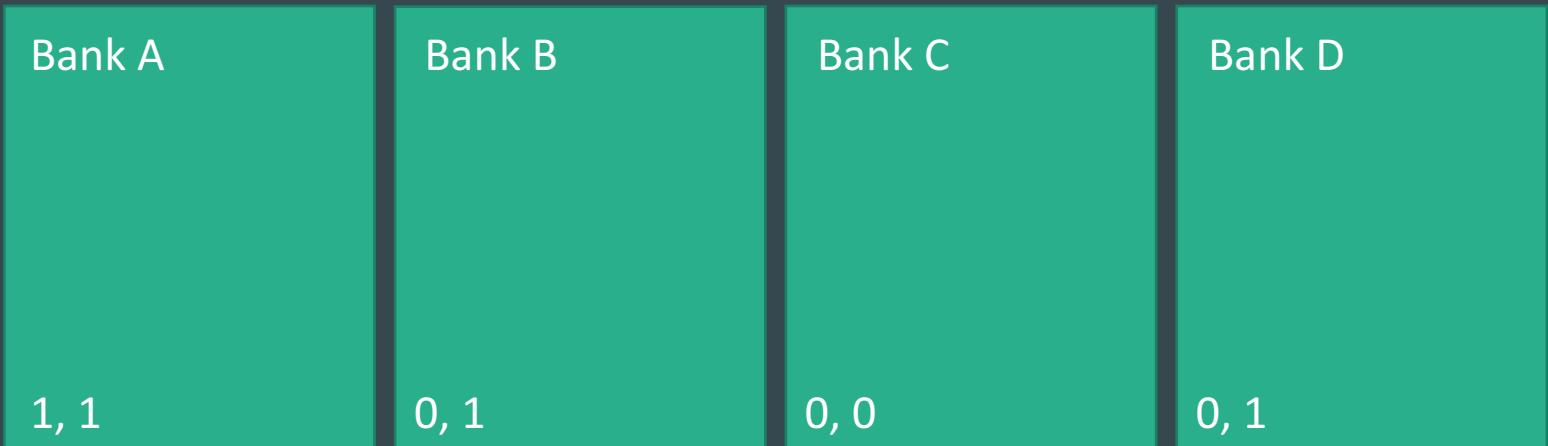
8, 13, 13, 12



$8 \rightarrow 1000_2$



$13 \rightarrow 1101_2$



# Planar memory layout



8, 13, 13, 12



$8 \rightarrow 1000_2$



$13 \rightarrow 1101_2$



$13 \rightarrow 1101_2$



$12 \rightarrow 1100_2$



# Original graphics format



Linear:

$10001101_2, 11011100_2, 11001101_2, 11011000_2$

Planar:

$11111111_2, 01111110_2, 00000000_2, 01100110_2$

# Decoding the game's graphics files

Required: Reading single bits

```
for (int bit = 7; bit >= 0; --bit) {  
    const auto bitMask = 1 << bit;  
    const auto valueOfBit = value & bitMask;  
    // work with valueOfBit  
}
```

# Decoding the game's graphics files

Wanted: Abstract away the bit-wrangling, write algorithm in terms of “sequence of values”

Solution: Create adapter iterator

# Adapter iterator: bit-wise reading

```
template<typename OriginalIterType>
class BitWiseIterator {
public:
    // some boilerplate omitted

    explicit BitWiseIterator(OriginalIterType originalIter) :
        mOriginalIter(originalIter), mBitIndex(0) {}

    // comparison operators, post-fix increment operator omitted

    BitWiseIterator& operator++();

    bool operator*() const;

private:
    OriginalIterType mOriginalIter;
    std::uint8_t mBitIndex;
};
```

# Adapter iterator: bit-wise reading

```
BitWiseIterator& operator++() {
    ++mBitIndex;
    if (mBitIndex == 8) {
        ++mOriginalIter;
        mBitIndex = 0;
    }
    return *this;
}

bool operator*() {
    const auto actualBitIndex = 7 - mBitIndex;
    const auto bitMask = 1 << actualBitIndex;
    const auto bitPack = *mOriginalIter;
    return (bitPack & bitMask) != 0;
}
```

# Adapter iterator: Client code

```
vector<uint8_t> decodedIndices(numPixels, 0);

BitWiseIterator<ByteBufferCIter> bitsIter(dataBegin);

for (auto plane = 0; plane < 4; ++plane) {
    auto destination = begin(decodedIndices);

    for (auto pixel = 0; pixel < pixelCount; ++pixel) {
        const auto planeBit = *bitsIter++;
        *destination++ |= planeBit << plane;
    }
}
```

# Enemy AI

# State machines for enemy AI: Example

Flying



Hovering



Plunging down



Rising up



# Red bird state machine



# "Old-school" approach

```
enum class BirdState {  
    Flying, Hovering, Plunging, Rising  
};  
  
struct RedBird {  
    BirdState mState;  
    int mOriginalHeight;  
    int mElapsedTime;  
};
```

```
switch (bird.mState) {  
case BirdState::Flying:  
    // Check if over player  
    break;  
  
case BirdState::Hovering:  
    // etc.
```

# Encoding states in a variant<>

```
struct Flying {};
struct Hovering { int mFramesElapsed = 0; };
struct PlungingDown { int mInitialHeight; };
struct RisingUp { int mInitialHeight; };

using StateT = variant<
    Flying, Hovering, PlungingDown, RisingUp>;

struct RedBird {
    StateT mState;
};
```

# Pattern matching on the state

```
atria::variant::match(birdState.mState,
 [&](const Flying&) {
    const auto wantsToAttack =
        position.y + 2 < playerPosition.y && position.x > playerPosition.x &&
        position.x < playerPosition.x + 2;

    if (wantsToAttack) {
        birdState.mState = Hovering{};
    }
 },

 [&](Hovering& state) {
    ++state.mFramesElapsed;
    if (state.mFramesElapsed >= 6) {
        birdState.mState = PlungingDown{position.y};
    }
 },

 // etc. for remaining states
```

# Other uses for variant pattern matching

- Script interpreter
  - Parse text into command objects
  - Script → `std::vector<variant<Command1, Command2, ...>>;`
- Game mode management (ingame vs. bonus-screen)
- Effect specification DSL

```
const EffectSpec SPIDER_KILL_EFFECT_SPEC[] = {
    {EffectSprite{{-1, 1}, 1, EffectMovement::None}, 0},
    {RandomExplosionSound{}, 0}
};

const EffectSpec RED_BIRD_KILL_EFFECT_SPEC[] = {
    {Particles{}, INGAME_PALETTE[5]}, 0,
    {EffectSprite{}, 1, EffectMovement::None}, 0,
    {RandomExplosionSound{}, 0}
};
```

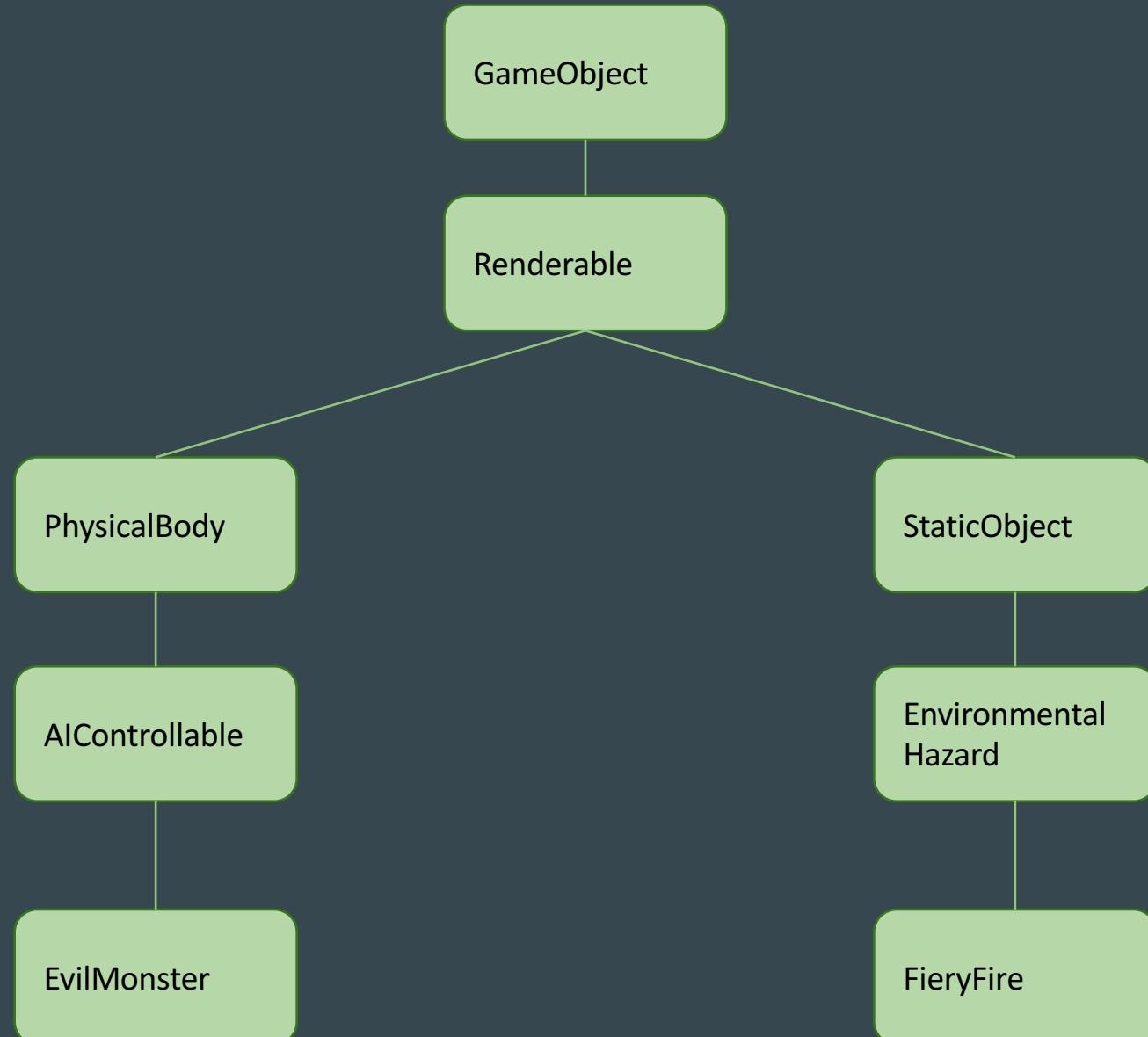
# Pattern matching in C++

- Library-based (<https://github.com/Ableton/atria>, others)
- [C++ Weekly Ep. 49](#): Inheriting lambdas to make a combined visitor
- Proposal to add native pattern matching to C++: [P0095](#)
- Rant about std::visit: <https://bitbashing.io/std-visit.html>

# Entity-Component-Systems

# Motivation: Provide more flexibility than OOP

```
class GameObject {  
public:  
    virtual void update(float timeDelta);  
};  
  
class Game {  
private:  
    vector<unique_ptr<GameObject>> mObjects;  
  
public:  
    void updateGame(const float timeDelta) {  
        for (auto& pObject : mObjects) {  
            pObject->update(timeDelta);  
        }  
    }  
};
```



“Can we have a monster  
that’s also on fire?”



# Entity, Component, System

## **Component**

Holds state, POD-like

## **Entity**

Handle/ID for a set of components

## **System**

Implements logic for combination of components

# Example components

Position

x, y, z

Renderable

vertices, textures

DamageInflictor

damageAmount,  
affectedGroups

BoundingBox

minX, minY, minZ,  
maxX, maxY, maxZ

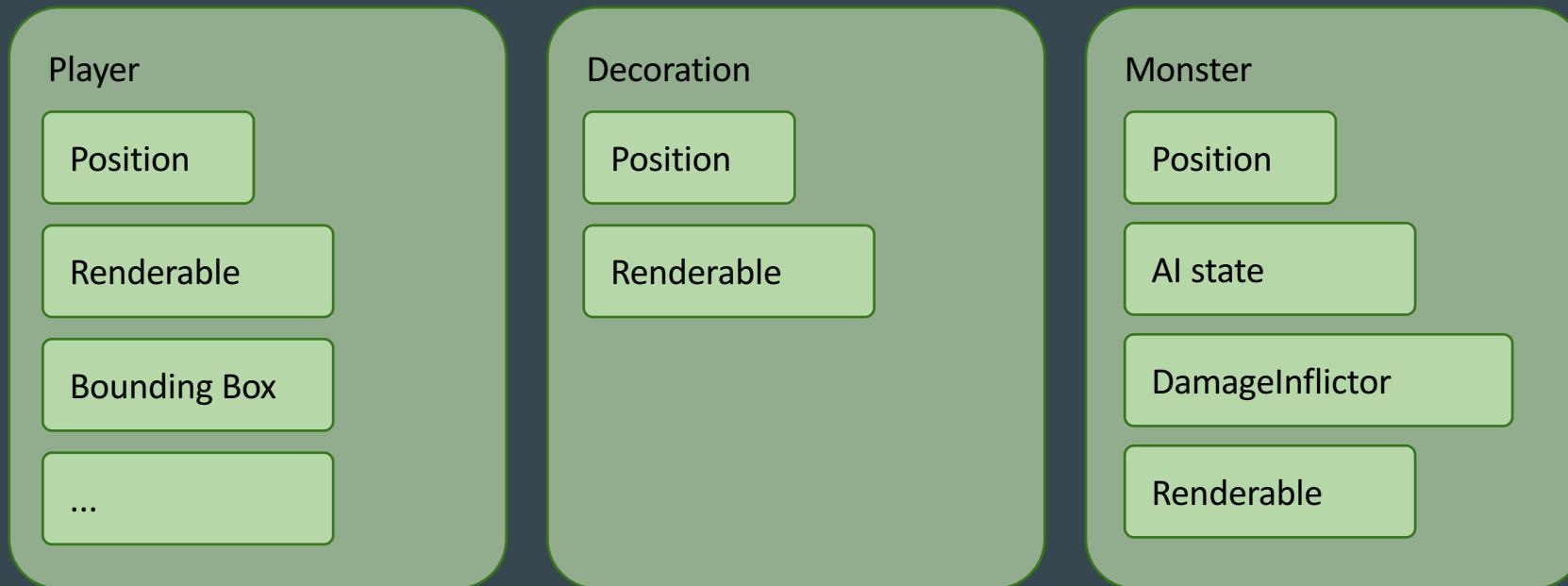
ItemPickup

givenItemType

PhysicalBody

velocity,  
acceleration

# Example entities



# Systems

Implement specific logic for certain combination of components

Can read and write components' data

Only operate on entities with required components

Can send and receive events from/to other systems/outside world

# Example systems

PhysicsSystem - needs Position, PhysicalBody

```
// update position component  
position += physicalBody.velocity;  
  
// update velocity  
physicalBody.velocity += physicalBody.acceleration * dt;
```

# Example systems

## CollisionDetectionSystem - needs Position, BoundingBox

```
const auto candidates = possibleColliders();
const auto it = find_if(begin(candidates), end(candidates),
    [&](const auto& collider) {
        return intersecting(collider.bbox, bbox);
});

if (it != end(candidates)) {
    emitEvent(Collision{*it, currentEntity});
}
```

# Example systems

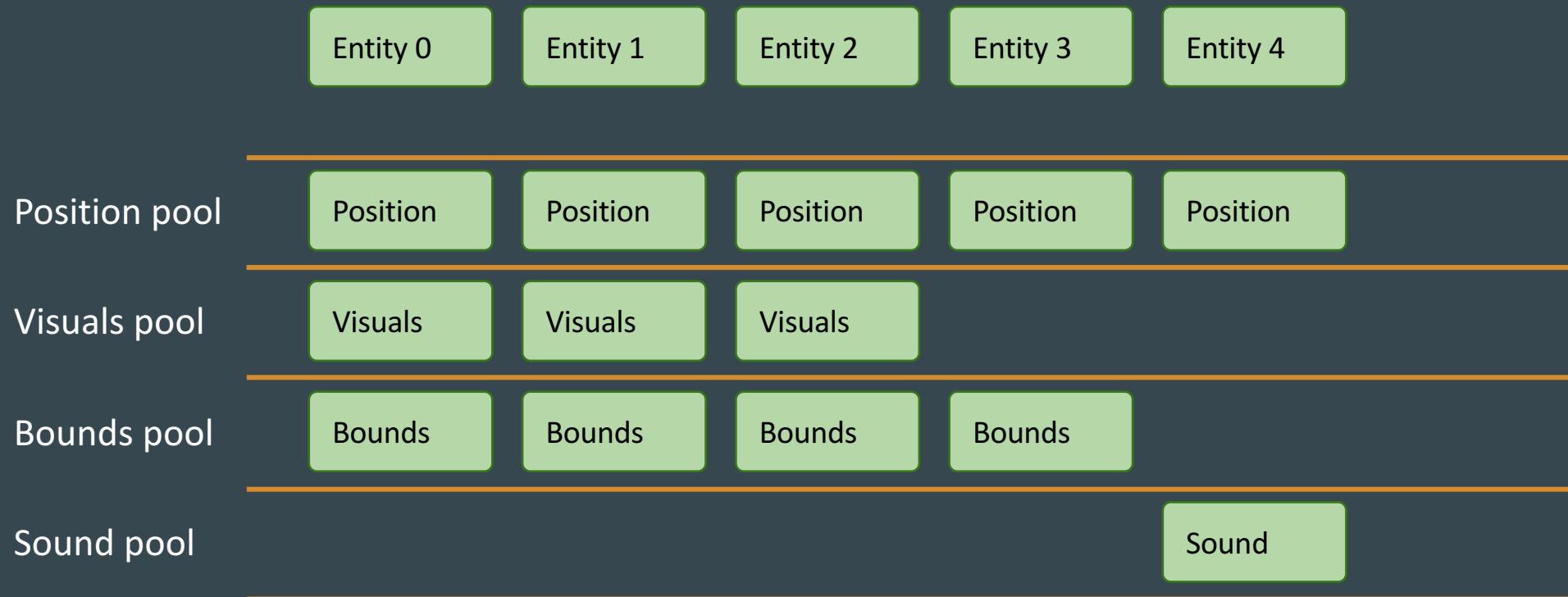
Rendering system – needs Renderable, Position, BoundingBox

AI controller system – needs AiState, Position

Sound emitter system – needs Sound, Position

etc.

# Component storage



# Composition at runtime

```
auto entity = entityManager.createEntity();

entity.addComponent<Position>(Position{3, 5});
entity.addComponent<AIState>(AIState{ai::Type::SmartMonster});

auto gameMap = loadMap();

for (const auto& description : gameMap.enemies()) {
    auto enemy = entityManager.createEntity();
    enemy.addComponent<Position>(description.position);
    enemy.addComponent<VisualRepresentation>(
        createVisualsForEnemyType(description.type));
}
```

# Behavior as combination of systems

Example: Projectile

- Movement handled by PhysicsSystem
- Collision checked by CollisionDetectionSystem
- Inflicts damage on enemy via DamageInflictionSystem
- Visual representation handled by RenderingSystem

# ECS libraries for C++

**EntityX** (<https://github.com/alecthomas/entityx>)

*A fast, type-safe C++ Entity-Component system*

**ECST** (<https://github.com/SuperV1234/ecst>)

*Experimental & work-in-progress C++14 multithreaded compile-time Entity-Component-System header-only library.*

**EnTT** (<https://github.com/skypjack/entt>)

*EnTT is a header-only, tiny and easy to use Entity-Component System in modern C++*

# ECS in Rigel Engine

```
case 19: // Rocket launcher
{
    CollectableItem item;
    item.mGivenScore = 2000;
    item.mGivenWeapon = WeaponType::Rocket;
    configureItemBox(
        entity,
        ContainerColor::Green,
        100,
        item);
}
break;

case 50: // teleporter
case 51: // teleporter
entity.assign<AnimationLoop>(1);
entity.assign<Interactable>(InteractableType::Teleporter);
entity.assign<BoundingBox>(BoundingBox{{2, 0}, {2, 5}});
break;
```

Let's have a look at the code!

# ECS: My experience

- Very easy to use (with a good library)
- Encourages code reuse/generic behaviors
- Encourages more “declarative” coding
- Explicit update order is very useful
  
- Solutions can be less obvious at first
- Needs some „getting used to”

Reverse Engineering  
the original

# Disassembling with Ida Pro

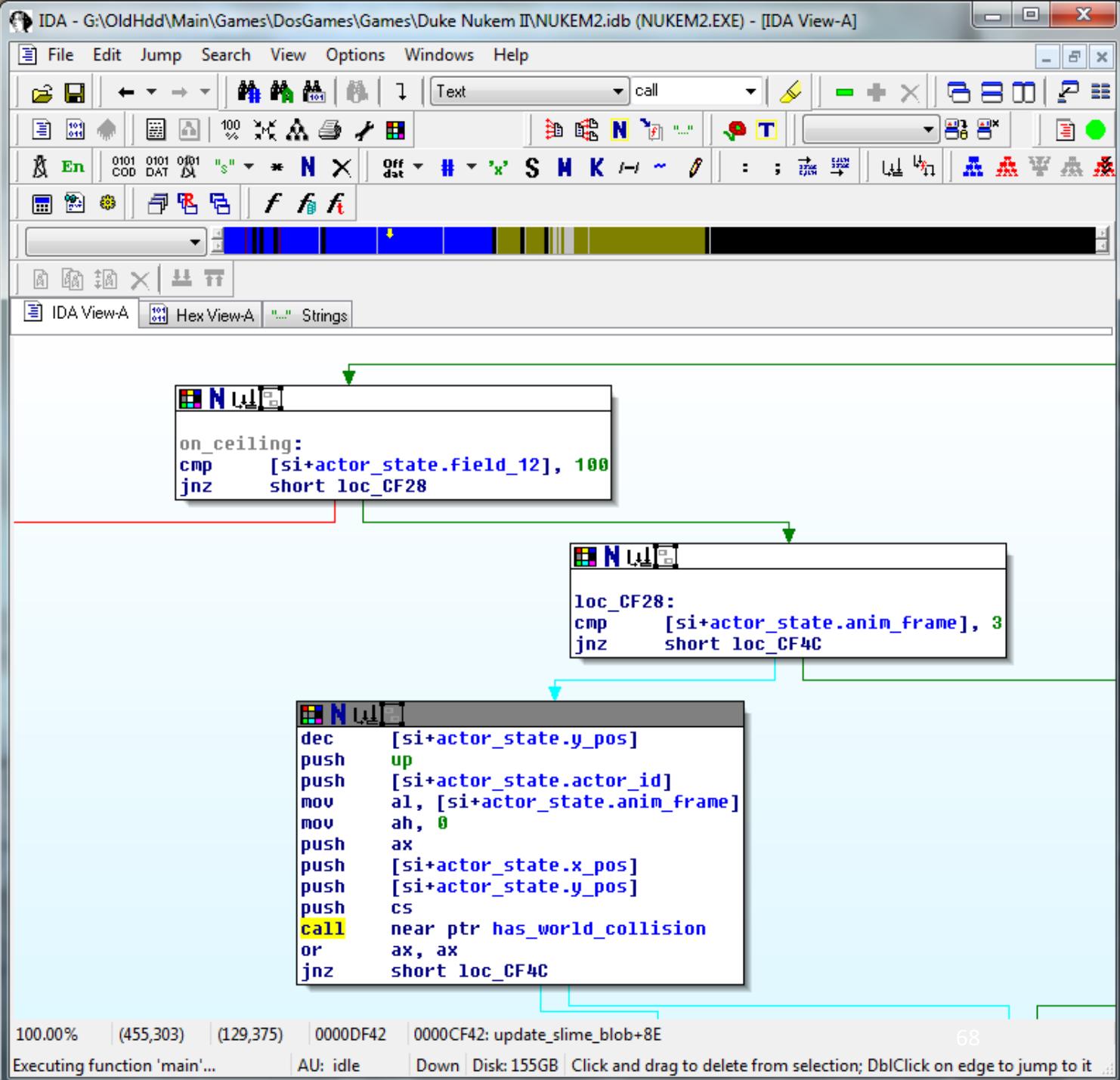
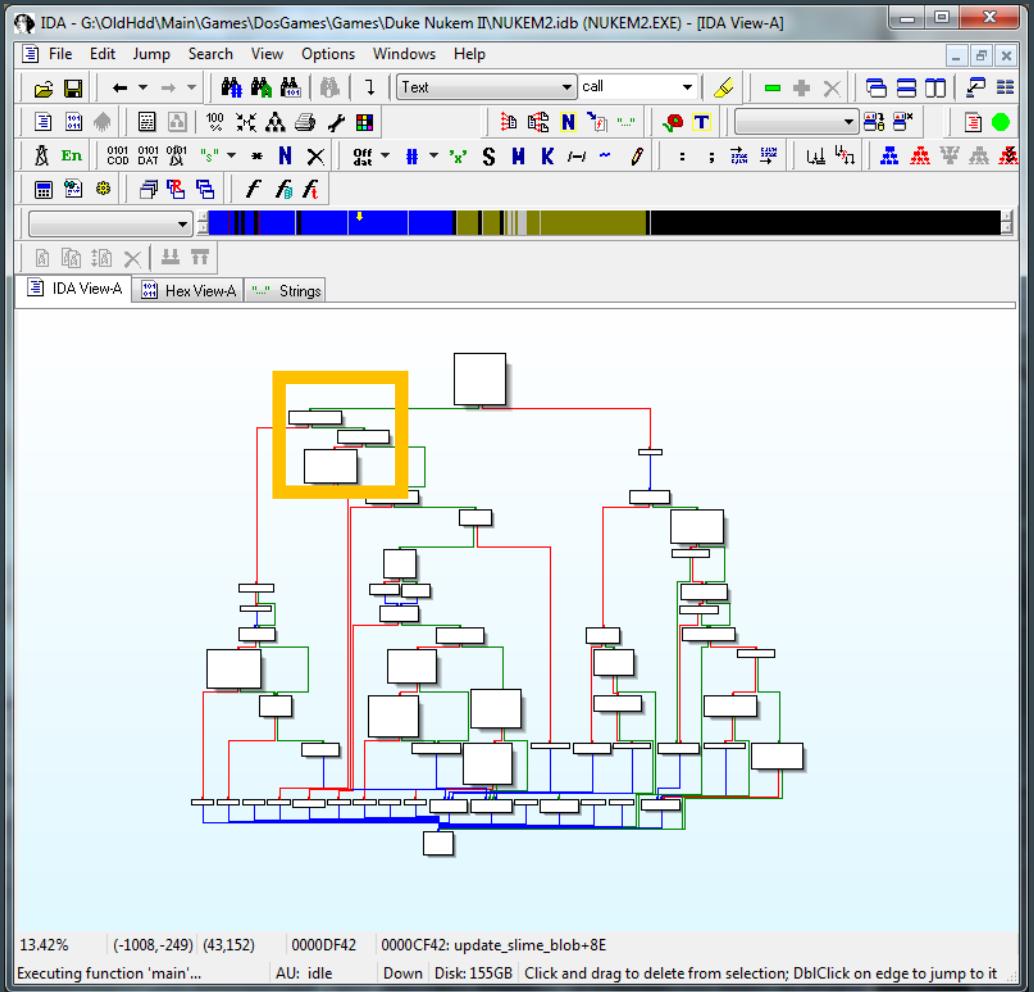
Ida Pro – The interactive disassembler

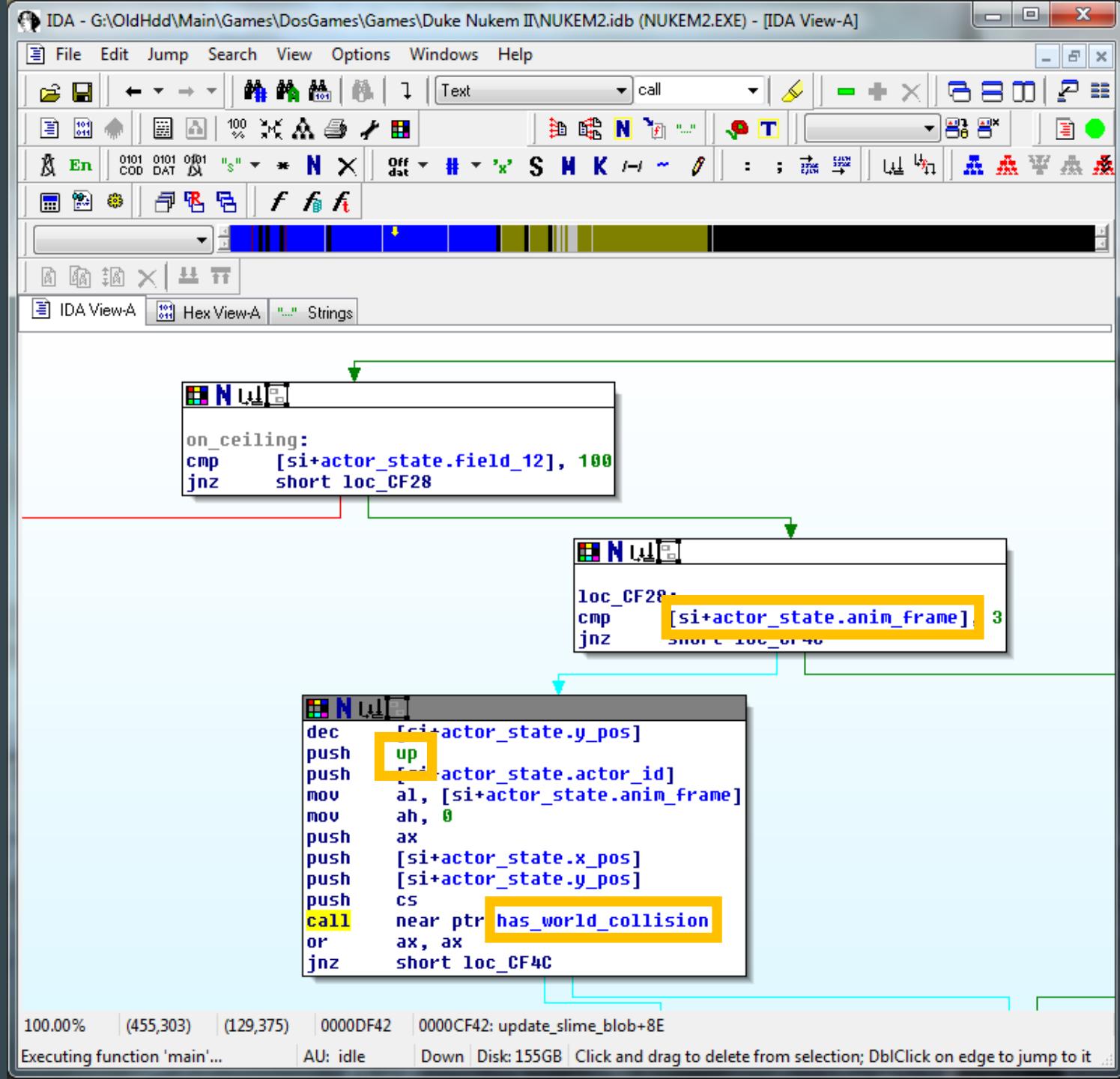
Features:

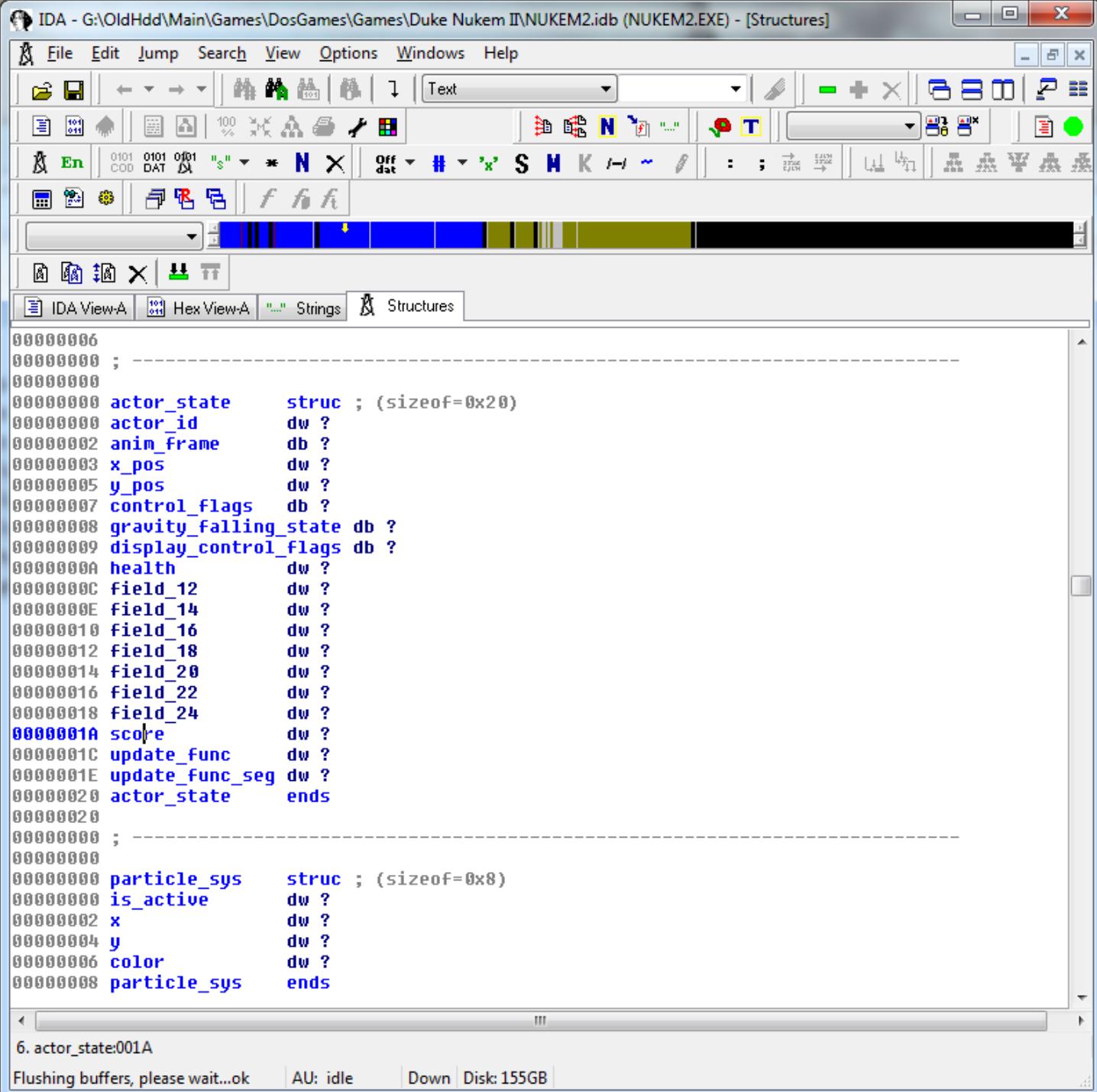
- Visualize control flow as graph
- Assign names to function parameters, variables, functions etc.
- Define struct layouts
- Jump to cross-references

Version 5 free for non-commercial use:

[https://www.hexrays.com/products/ida/support/download\\_freeware.shtml](https://www.hexrays.com/products/ida/support/download_freeware.shtml)

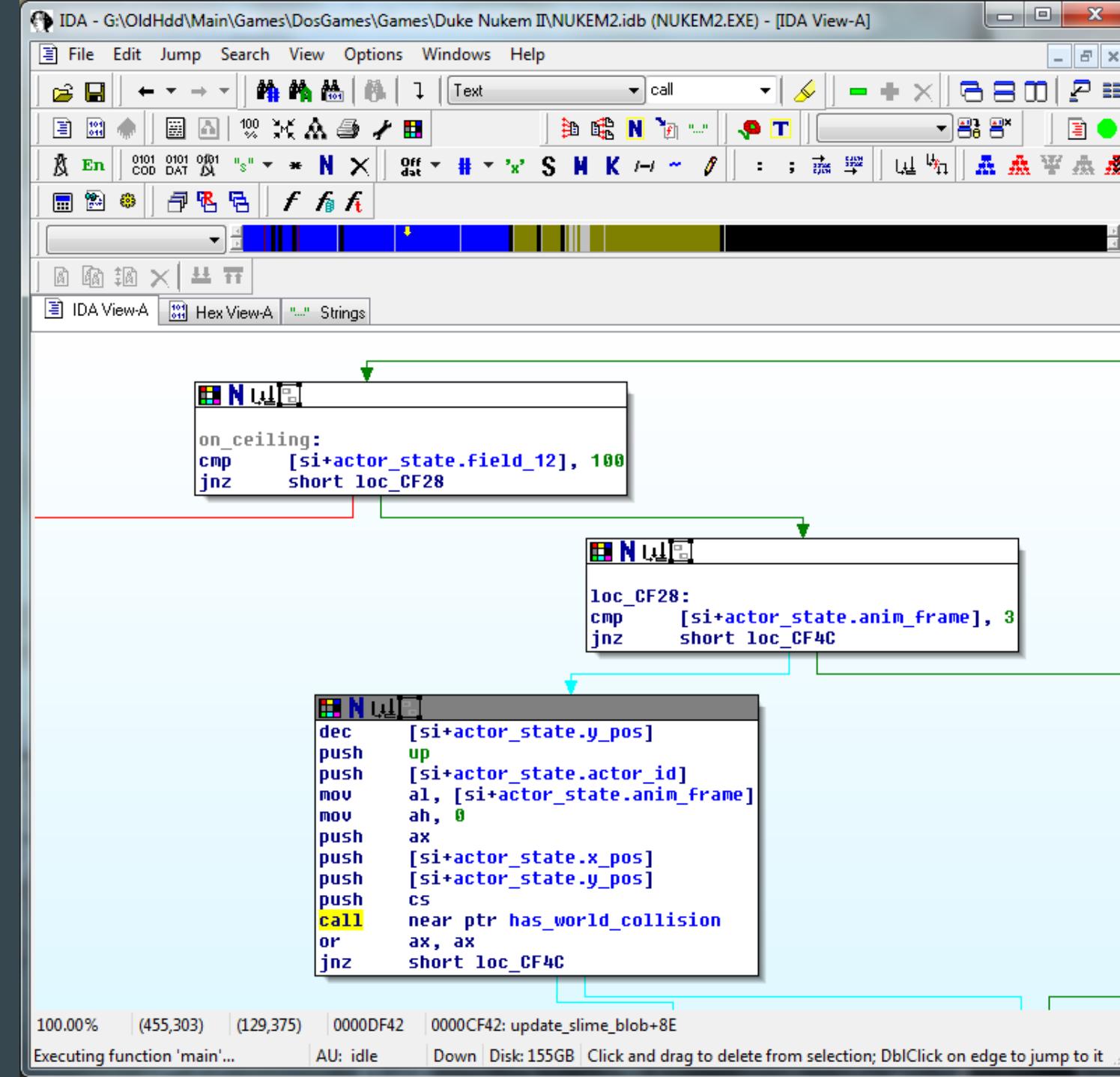






# Disassembling: Workflow

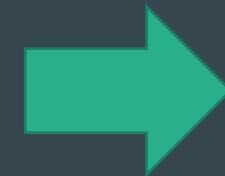
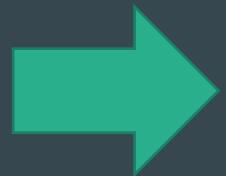
- Open graph for function
- Make sure known names/definitions etc. are applied
- Transcribe assembly into C-like pseudo-code
- Simplify pseudo-code, identify underlying logic
- C++ implementation



```
const int UP = ...;

if (state->field_12 != 100) {
    if (state->anim_frame == 3) {
        --state->y_pos;
        if (has_world_collision(
            UP,
            state->actor_id,
            state->anim_frame,
            state->x_pos,
            state->y_pos)
        ) {
            // ...
        }
    }
}
```

# Stepping through video captures



```
ffmpeg -i nukem2_051.avi -r 17.5 f%000d.png
```

# Stepping through video captures



# Funny hacks and trivia

# (Not so random) random numbers

```
int random_number() {
    static uint8_t index = 0;
    ++index;
    return RANDOM_NUMBER_TABLE[index];
}
```

# (Not so random) random numbers

```
const std::array<int, 256> RANDOM_NUMBER_TABLE{
    0, 8, 109, 220, 222, 241, 149, 107, 75, 248, 254, 140, 16, 66, 74, 21, 211, 47,
    80, 242, 154, 27, 205, 128, 161, 89, 77, 36, 95, 110, 85, 48, 212, 140, 211,
    249, 22, 79, 200, 50, 28, 188, 52, 140, 202, 120, 68, 145, 62, 70, 184, 190,
    91, 197, 152, 224, 149, 104, 25, 178, 252, 182, 202, 182, 141, 197, 4, 81, 181,
    242, 145, 42, 39, 227, 156, 198, 225, 193, 219, 93, 122, 175, 249, 0, 175, 143,
    70, 239, 46, 246, 163, 53, 163, 109, 168, 135, 2, 235, 25, 92, 20, 145, 138,
    77, 69, 166, 78, 176, 173, 212, 166, 113, 94, 161, 41, 50, 239, 49, 111, 164,
    70, 60, 2, 37, 171, 75, 136, 156, 11, 56, 42, 146, 138, 229, 73, 146, 77, 61,
    98, 196, 135, 106, 63, 197, 195, 86, 96, 203, 113, 101, 170, 247, 181, 113, 80,
    250, 108, 7, 255, 237, 129, 226, 79, 107, 112, 166, 103, 241, 24, 223, 239,
    120, 198, 58, 60, 82, 128, 3, 184, 66, 143, 224, 145, 224, 81, 206, 163, 45,
    63, 90, 168, 114, 59, 33, 159, 95, 28, 139, 123, 98, 125, 196, 15, 70, 194,
    253, 54, 14, 109, 226, 71, 17, 161, 93, 186, 87, 244, 138, 20, 52, 123, 251,
    26, 36, 17, 46, 52, 231, 232, 76, 31, 221, 84, 37, 216, 165, 212, 106, 197,
    242, 98, 43, 39, 175, 254, 145, 190, 84, 118, 222, 187, 136, 120, 163, 236, 249
};
```

# Clock deviation due to timer interrupt

Timer interrupt (INT 8): [https://en.wikipedia.org/wiki/Intel\\_8253](https://en.wikipedia.org/wiki/Intel_8253)

Regular frequency: 18.2 Hz

Default handler: Advance time-of-day, time-out floppy disk drive, ...

Game reprograms timer to approx. 280 Hz

invokes original handler every 15th time

$$280 / 18.2 \approx 15,385$$

$$280 / 15 \approx 18.6 \rightarrow \text{clock runs } \approx 2.5\% \text{ faster} \rightarrow 10 \text{ hrs game} \approx 15 \text{ min offset}$$

Thanks for listening!

<https://github.com/lethal-guitar/RigelEngine>