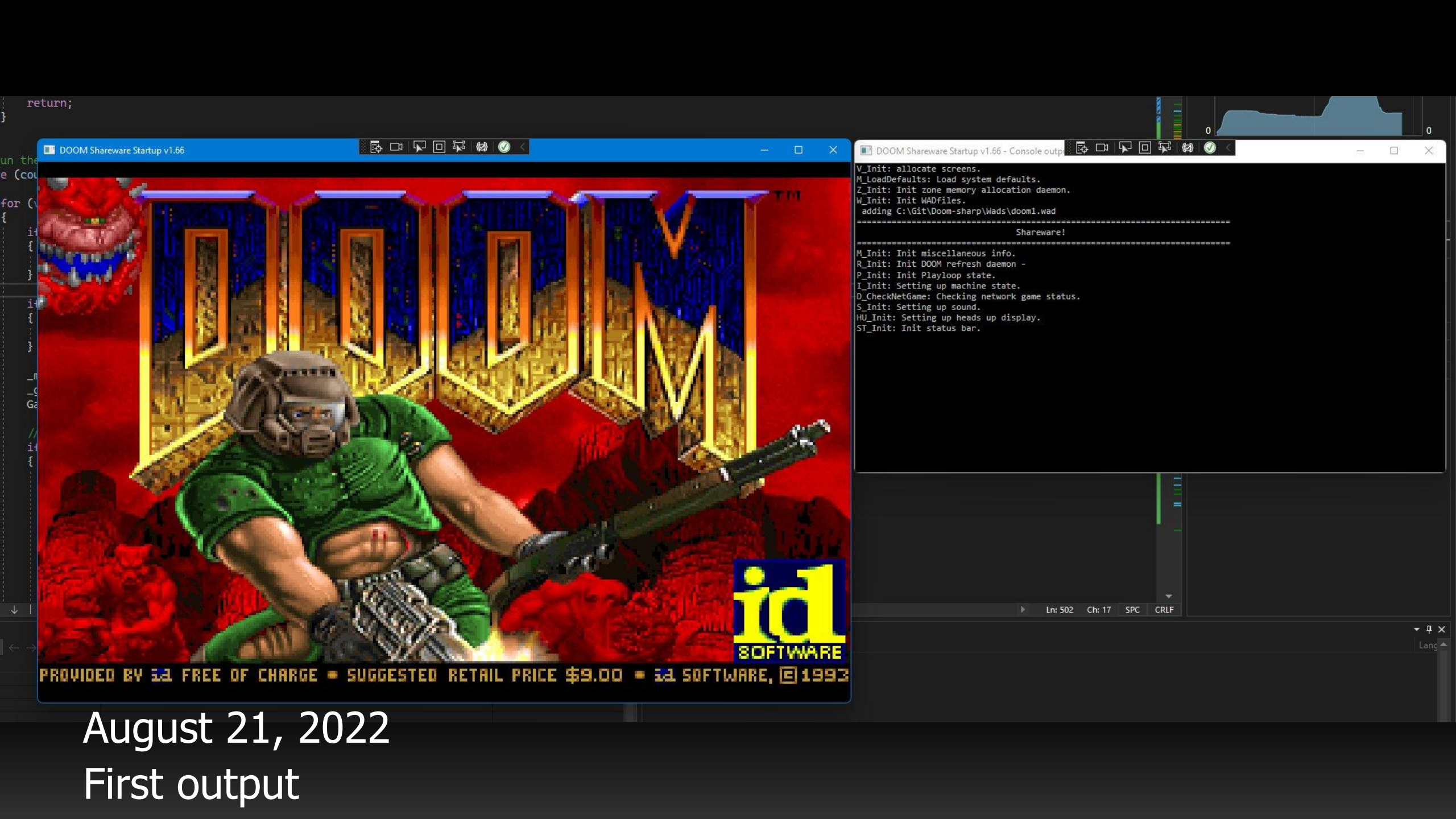
Blasting demons using C# without the unsafe keyword

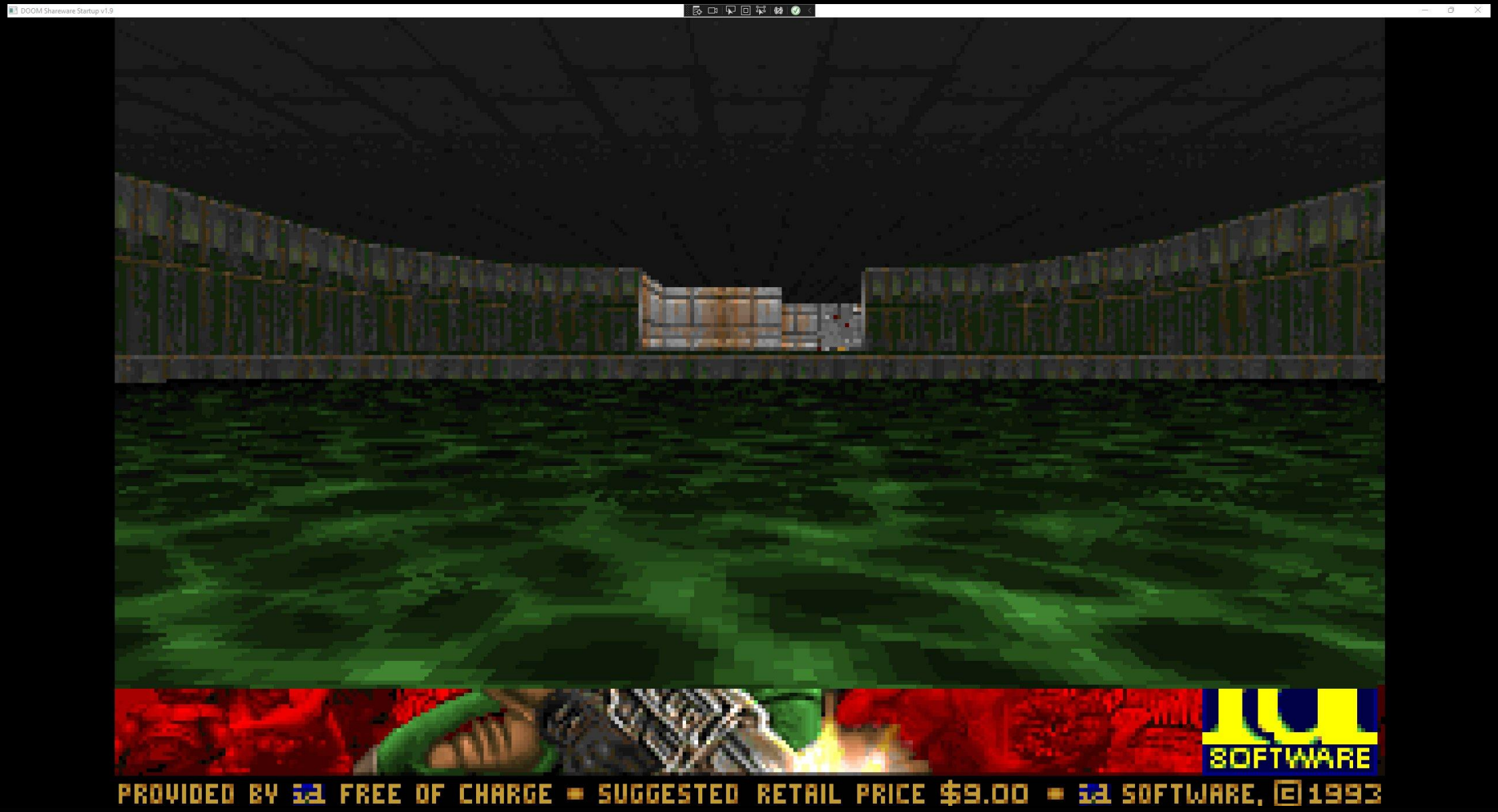
Hi, I’m Wesley

A Coding Architect at Xebia, coffee enthusiast, beer aficionado, and I have some crazy side projects.



August 21, 2022

First output

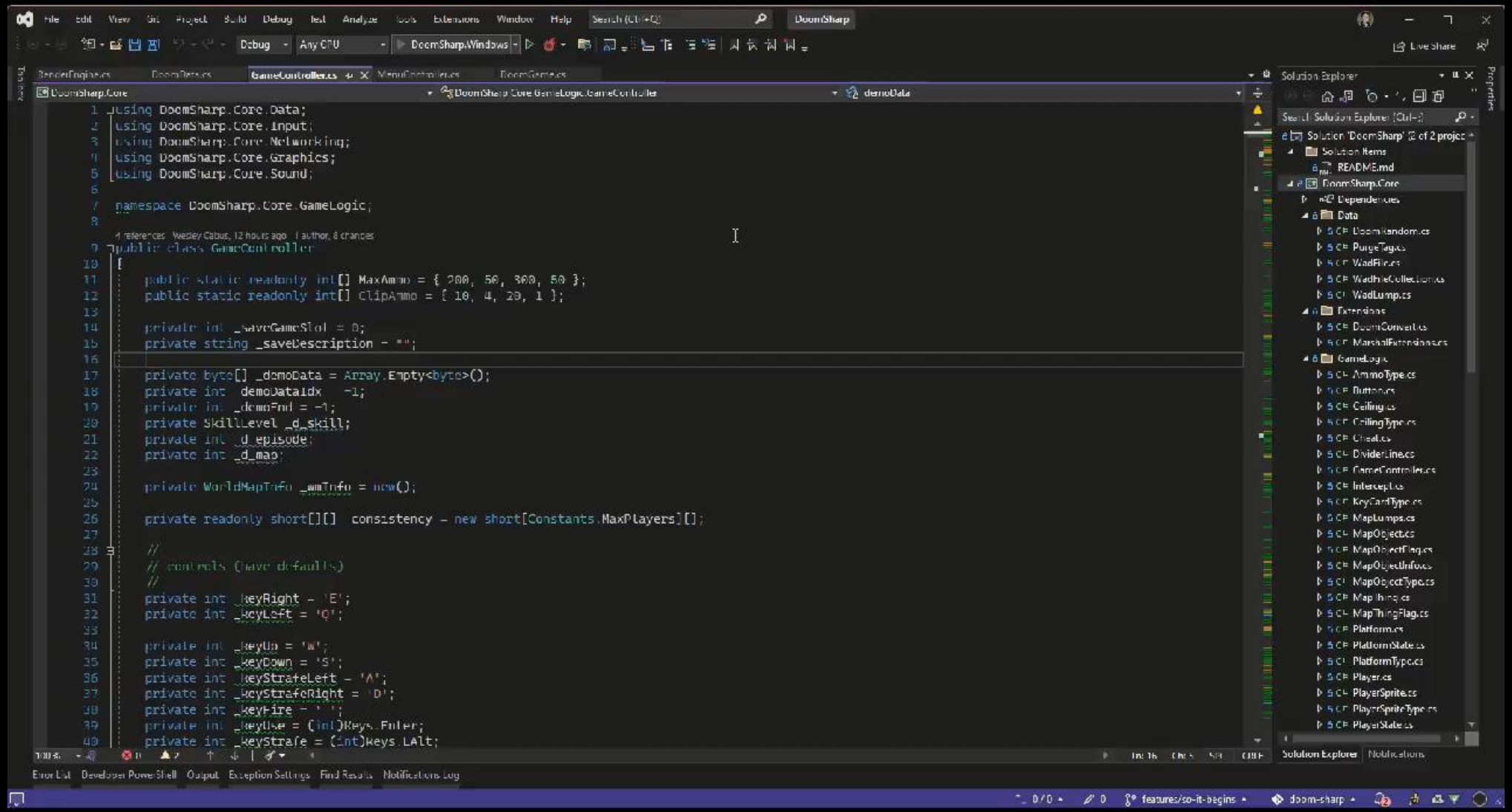


August 29, 2022

Rendering mostly works, missing status bar and input handling

August 30, 2022

Added the status bar



September 1, 2022

The pogo-stick incident (added input handling & player actions)

September 11, 2022

Rendering and movement bugs fixed, some items and areas react

September 25, 2022

80% finished! (sound and music was added later)

The unsafe keyword

The unsafe keyword

Unsafe context

- Code blocks
- Methods
- Types

Direct memory access

- Pointers
- Allocate and free memory

```
int[] a = [10, 20, 30, 40, 50];

// Must be in unsafe code to use interior pointers.
unsafe
{
    // Pin object on heap so that it doesn't move while using interior pointers.
    fixed (int* p = &a[0])
    {
        // p is pinned as well as object, so create another pointer
        // to show incrementing it.
        int* p2 = p;
        Console.WriteLine(*p2);
        // Incrementing p2 bumps the pointer by four bytes due to its type ...
        p2 += 1;
        Console.WriteLine(*p2);
        p2 += 1;
        Console.WriteLine(*p2);
        Console.WriteLine("--------");
        Console.WriteLine(*p);
        // Dereferencing p and incrementing changes the value of a[0] ...
        *p += 1;
        Console.WriteLine(*p);
        *p += 1;
        Console.WriteLine(*p);
    }
}
```

```
int[] a = [10, 20, 30, 40, 50];

unsafe
{
    fixed (int* p = &a[0])
    {
        int* p2 = p;
        Console.WriteLine(*p2);
        p2 += 1;

        Console.WriteLine(*p);
        *p += 1;
        Console.WriteLine(*p);
        *p += 1;
        Console.WriteLine(*p);
    }
}
```

```
int[] a = [10, 20, 30, 40, 50];

unsafe
{
    fixed (int* p = &a[0])
    {
        int* p2 = p;
        Console.WriteLine(*p2);
        p2 += 1;

        Console.WriteLine(*p);
        *p += 1;
        Console.WriteLine(*p);
        *p += 1;
        Console.WriteLine(*p);
    }
}
```

```
int[] a = [10, 20, 30, 40, 50];

unsafe
{
    fixed (int* p = &a[0])
    {
        int* p2 = p;
        Console.WriteLine(*p2);
        p2 += 1;

        Console.WriteLine(*p);
        *p += 1;
        Console.WriteLine(*p);
        *p += 1;
        Console.WriteLine(*p);
    }
}
```

```
int[] a = [10, 20, 30, 40, 50];

unsafe
{
    fixed (int* p = &a[0])
    {
        int* p2 = p;
        Console.WriteLine(*p2);
        p2 += 1;

        Console.WriteLine(*p);
        *p += 1;
        Console.WriteLine(*p);
        *p += 1;
        Console.WriteLine(*p);
    }
}
```

```
int[] a = [10, 20, 30, 40, 50];

unsafe
{
    fixed (int* p = &a[0])
    {
        int* p2 = p;
        Console.WriteLine(*p2);
        p2 += 1;

        Console.WriteLine(*p);
        *p += 1;
        Console.WriteLine(*p);
        *p += 1;
        Console.WriteLine(*p);
    }
}
```

```
int[] a = [10, 20, 30, 40, 50];

unsafe
{
    fixed (int* p = &a[0])
    {
        int* p2 = p;
        Console.WriteLine(*p2);
        p2 += 1;

        Console.WriteLine(*p);
        *p += 1;
        Console.WriteLine(*p);
        *p += 1;
        Console.WriteLine(*p);
    }
}
```

Unsafe doesn’t mean that the code is not safe. Its safety just can’t be automatically verified.


Here be dragons

Benchmark: unsafe versus safe

```
int[] a = [10, 20, 30, 40, 50];
unsafe
{
    fixed (int* p = &a[0])
    {
        int* p2 = p;
        Console.WriteLine(*p2);
        p2 += 1;
        Console.WriteLine(*p2);
        p2 += 1;
        Console.WriteLine(*p2);
        Console.WriteLine("--------");
        Console.WriteLine(*p);
        *p += 1;
        Console.WriteLine(*p);
        *p += 1;
        Console.WriteLine(*p);
    }
}
```

```
int[] a = [10, 20, 30, 40, 50];


var index = 0;

var index2 = index;
Console.WriteLine(a[index2]);
index2 += 1;
Console.WriteLine(a[index2]);
index2 += 1;
Console.WriteLine(a[index2]);
Console.WriteLine("--------");
Console.WriteLine(a[index]);
a[index] += 1;
Console.WriteLine(a[index]);
a[index] += 1;
Console.WriteLine(a[index]);
```

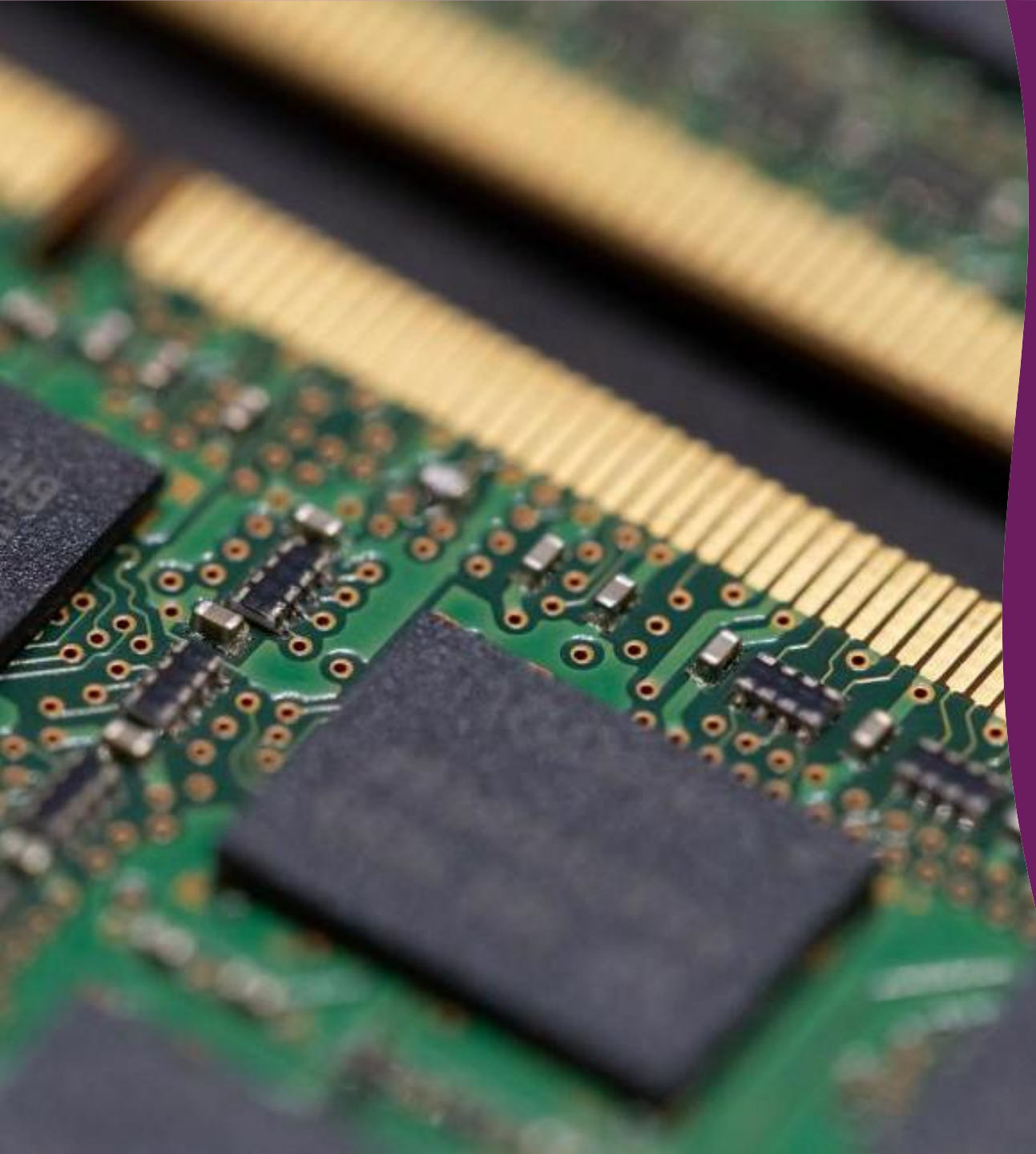
Benchmark: unsafe versus safe

Let’s look at the intermediate language

Benchmark: unsafe versus safe

Runtime	Method	Mean	Allocated
.NET 6.0	Unsafe	800,0 us	273 B
.NET 6.0	Safe	819,5 us	273 B
.NET 7.0	Unsafe	798,0 us	273 B
.NET 7.0	Safe	799,8 us	273 B
.NET 8.0	Unsafe	805,7 us	48 B
.NET 8.0	Safe	789,3 us	48 B

1 us = 1 microsecond = 0,000001 second



Memory management

Memory management

- int *pointer
- &address
- malloc(), calloc(), realloc()
- free()
- new, delete

Memory management – malloc/free

```
void sample(int nameLength) {
    char name[100];
    char* name2 = malloc(nameLength * sizeof(char));
    strcpy(name2, "Test");

    int weird = (int)(*(name2 + 3));
    printf("Weird = %d", weird); // Weird = 116

    free(name2);
}
```

Memory management – malloc/free

```
void sample(int nameLength) {
    char name[100];
    char* name2 = malloc(nameLength * sizeof(char));
    strcpy(name2, "Test");

    int weird = (int)(*(name2 + 3));
    printf("Weird = %d", weird); // Weird = 116

    free(name2);
}
```

Memory management – malloc/free

```
void sample(int nameLength) {
    char name[100];
    char* name2 = malloc(nameLength * sizeof(char));
    strcpy(name2, "Test");

    int weird = (int)(*(name2 + 3));
    printf("Weird = %d", weird); // Weird = 116

    free(name2);
}
```

Memory management – malloc/free

```
void sample(int nameLength) {
    char name[100];
    char* name2 = malloc(nameLength * sizeof(char));
    strcpy(name2, "Test");

    int weird = (int)(*(name2 + 3));
    printf("Weird = %d", weird); // Weird = 116

    free(name2);
}
```

Memory management – malloc/free

```
void sample(int nameLength) {
    char name[100];
    char* name2 = malloc(nameLength * sizeof(char));
    strcpy(name2, "Test");

    int weird = (int)(*(name2 + 3));
    printf("Weird = %d", weird); // Weird = 116

    free(name2);
}
```

Memory management – new/delete

```
#include <print>

typedef struct {
    char name[100];
    int age;
} Person;

void sample2() {
    const Person *person = new Person();
    int age = person->age;
    std::print("Less weird = {}", age);
    delete person;
}
```

Memory management – z_zone.c

```
int mb_used = 6;
byte* I_ZoneBase (int* size)
{
    *size = mb_used*1024*1024;
    return (byte *) malloc (*size);
}

void Z_Init (void)
{
    memblock_t* block;
    int size;

    mainzone = (memzone_t *)I_ZoneBase (&size);
    // ...
```

Memory management – z_zone.c

```
int mb_used = 6;
byte* I_ZoneBase (int* size)
{
    *size = mb_used*1024*1024;
    return (byte *) malloc (*size);
}

void Z_Init (void)
{
    memblock_t* block;
    int size;

    mainzone = (memzone_t *)I_ZoneBase (&size);
    // ...
```

Memory management – z_zone.c

```
int mb_used = 6;
byte* I_ZoneBase (int* size)
{
    *size = mb_used*1024*1024;
    return (byte *) malloc (*size);
}

void Z_Init (void)
{
    memblock_t* block;
    int size;

    mainzone = (memzone_t *)I_ZoneBase (&size);
    // ...
```

Memory management – using Z_Malloc

```
void P_LoadVertexes (int lump)
{
    byte* data;
    int i;
    mapvertex_t* ml;
    vertex_t* li;

    // Determine number of lumps:
    //  total lump length / vertex record length.
    numvertexes = W_LumpLength (lump) / sizeof(mapvertex_t);

    // Allocate zone memory for buffer.
    vertexes = Z_Malloc (numvertexes*sizeof(vertex_t),PU_LEVEL,0);
```

Memory management – Using arrays instead

```
private void P_LoadVertexes(int lump)
{
    // Determine number of lumps:
    //  total lump length / vertex record length.
    _numVertices = DoomGame.Instance.WadData.LumpLength(lump) / 4; // two shorts

    // Allocate zone memory for buffer.
    _vertices = new Vertex[_numVertices];

    // Load data into cache.
    var data = DoomGame.Instance.WadData.GetLumpNum(lump, PurgeTag.Static)!;

    // ...
```

Memory management – Using arrays instead

```
private void P_LoadVertexes(int lump)
{
    // Determine number of lumps:
    //  total lump length / vertex record length.
    _numVertices = DoomGame.Instance.WadData.LumpLength(lump) / 4; // two shorts

    // Allocate zone memory for buffer.
    _vertices = new Vertex[_numVertices];

    // Load data into cache.
    var data = DoomGame.Instance.WadData.GetLumpNum(lump, PurgeTag.Static)!;

    // ...
```

List<T> vs T[ ]

Let’s do another benchmark

- Pre-allocated array
- Pre-allocated list
- Resized list
- Double sequence resized array
- Chunked resized array
- Aggressively resized array

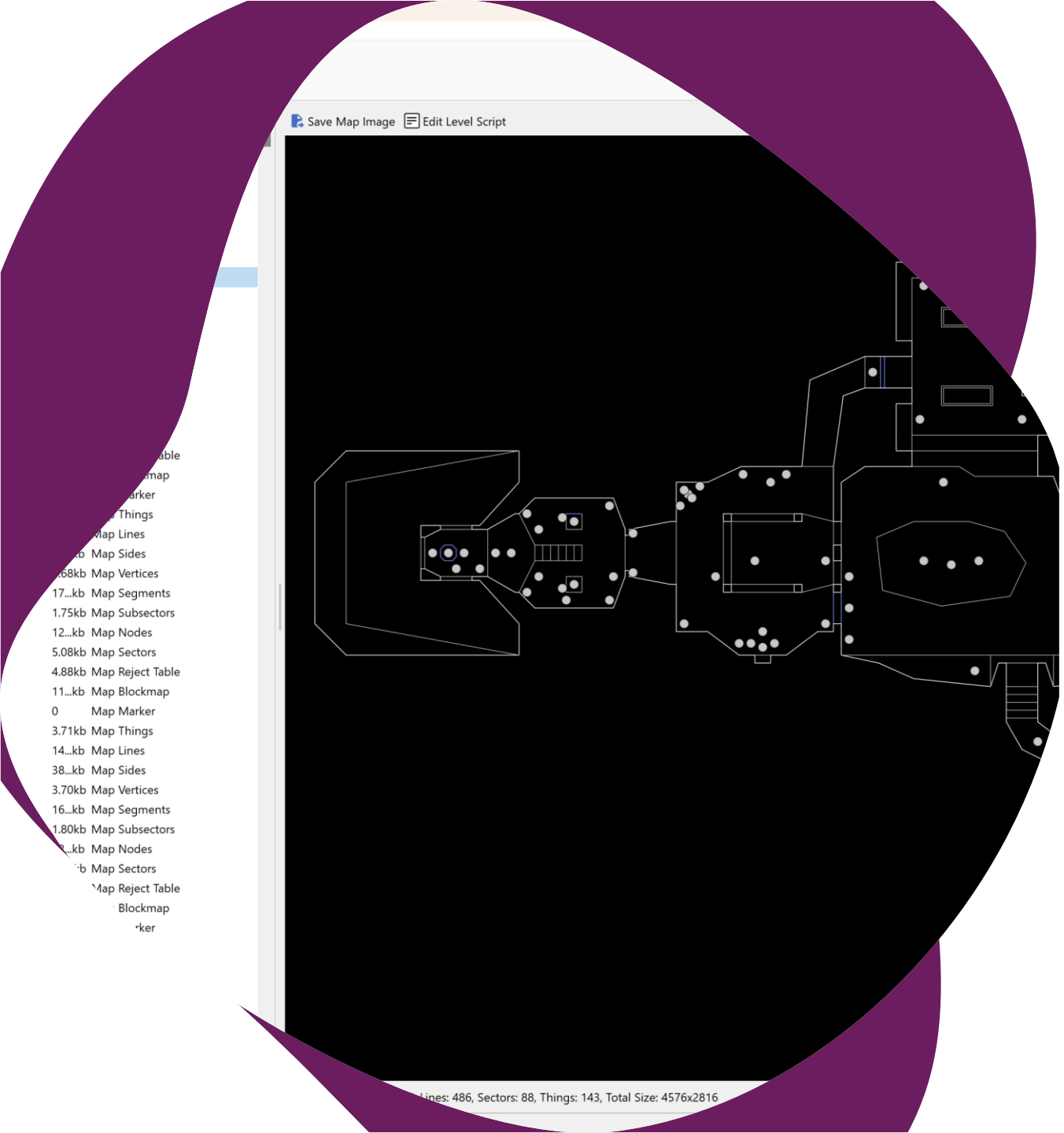
Benchmark: List<T> vs T[ ]

Results only show .NET 8

Method	Mean	Gen 0	Gen 1	Allocated
PreAllocatedArray	6,789 us	0,3281	-	4,02 KB
PreAllocatedList	7,298 us	0,3281	-	4,05 KB
DynamicallyResizedList	8,083 us	0,6714	-	8,23 KB
DynamicallyResizedDoublingSequence	7,453 us	0,6561	-	8,20 KB
DynamicallyResizedChunked	17,919 us	17,5781	0,1526	215,66 KB
DynamicallyResizedAggressive	98,730 us	169,4336	1,7090	2076 KB

1 us = 1 microsecond = 0,000001 second

Binary data



Binary data – WAD file

- Where's All the Data
- Header
- TOC / directory
- Lumps containing game data

Binary data – Reading files

```
void W_AddFile (char *filename)
{
    wadinfo_t header;
    int handle;
    // open the file and add to directory
    if ( (handle = open (filename,O_RDONLY | O_BINARY)) == -1)
    {
        printf (" couldn't open %s\n",filename);
        return;
    }
    printf (" adding %s\n",filename);
    // ...
    read (handle, &header, sizeof(header));
```

Binary data – Reading files

```
void W_AddFile (char *filename)
{
    wadinfo_t header;
    int handle;
    // open the file and add to directory
    if ( (handle = open (filename,O_RDONLY | O_BINARY)) == -1)
    {
        printf (" couldn't open %s\n",filename);
        return;
    }
    printf (" adding %s\n",filename);
    // ...
    read (handle, &header, sizeof(header));
```

Binary data – Reading files

```
void W_AddFile (char *filename)
{
    wadinfo_t header;
    int handle;
    // open the file and add to directory
    if ( (handle = open (filename,O_RDONLY | O_BINARY)) == -1)
    {
        printf (" couldn't open %s\n",filename);
        return;
    }
    printf (" adding %s\n",filename);
    // ...
    read (handle, &header, sizeof(header));
```

Binary data – Reading files

```
void W_AddFile (char *filename)
{
    wadinfo_t header;
    int handle;
    // open the file and add to directory
    if ( (handle = open (filename,O_RDONLY | O_BINARY)) == -1)
    {
        printf (" couldn't open %s\n",filename);
        return;
    }
    printf (" adding %s\n",filename);
    // ...
    read (handle, &header, sizeof(header));
```

Binary data – Reading files

```
public Task<WadFile?> LoadFromFile(string file)
{
    var fs = new FileStream(file, FileMode.Open, FileAccess.Read, FileShare.None);
    var br = new BinaryReader(fs, Encoding.ASCII, leaveOpen: false);

    DoomGame.Console.WriteLine($" adding {file}");

    if (string.Equals(Path.GetExtension(file), ".wad",
        StringComparison.OrdinalIgnoreCase))
    {
        // WAD file
        return Task.FromResult(LoadWad(file, br));
    }
    return Task.FromResult<WadFile?>(null);
}
```

Binary data – Reading files

```
private static WadFile? LoadWad(string file, BinaryReader reader)
{
    var header = new WadFile.WadInfo
    {
        Identification = Encoding.ASCII.GetString(reader.ReadBytes(4)).TrimEnd('\0'),
        NumLumps = reader.ReadInt32(),
        InfoTableOfs = reader.ReadInt32()
    };

    var wadFile = new WadFile(reader) { Header = header };

    // ...

    return wadFile;
}
```

Binary data – Reading files

```
private static WadFile? LoadWad(string file, BinaryReader reader)
{
    var header = new WadFile.WadInfo
    {
        Identification = Encoding.ASCII.GetString(reader.ReadBytes(4)).TrimEnd('\0'),
        NumLumps = reader.ReadInt32(),
        InfoTableOfs = reader.ReadInt32()
    };

    var wadFile = new WadFile(reader) { Header = header };

    // ...

    return wadFile;
}
```

Binary data – Reading files

```
private static WadFile? LoadWad(string file, BinaryReader reader)
{
    var header = new WadFile.WadInfo
    {
        Identification = Encoding.ASCII.GetString(reader.ReadBytes(4)).TrimEnd('\0'),
        NumLumps = reader.ReadInt32(),
        InfoTableOfs = reader.ReadInt32()
    };

    var wadFile = new WadFile(reader) { Header = header };

    // ...

    return wadFile;
}
```

WAD – File Lumps

```
typedef struct
{
    int filepos;
    int size;
    char name[8];
} filelump_t;
```

- What is the size of this structure?
- How many bytes is a “char” in the system?
- How many bytes is an “int” in the system?
- filelump_t => 16 bytes

Endianness – 0x4A3B2C1D

Big-endian

Most-significant byte first

4A 3B 2C 1D

Motorola 68000, SPARC, mainframes

Little-endian

Least-significant byte first

1D 2C 3B 4A

MOS 6502, DEC VAX, **Intel x86**

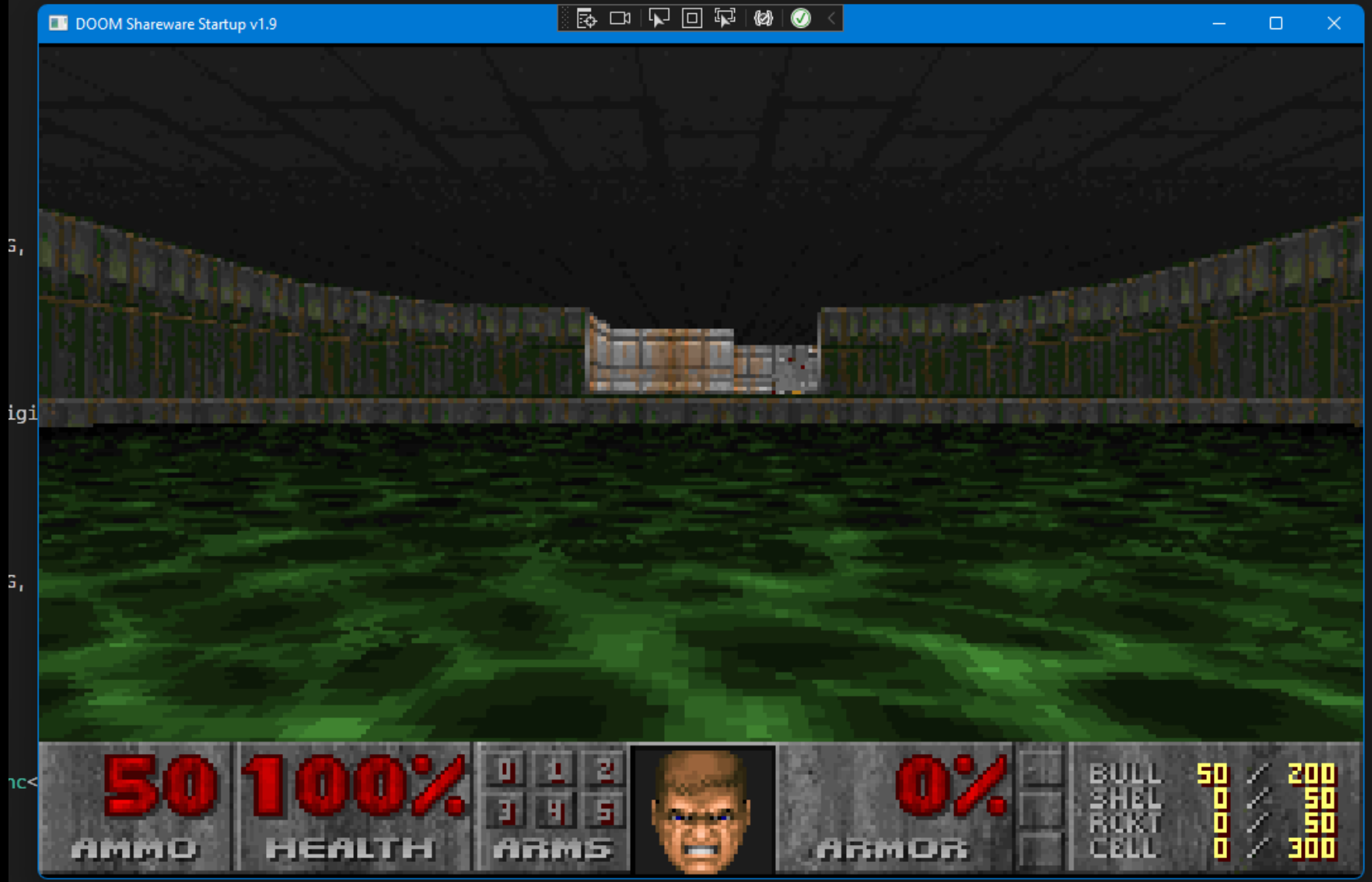
Middle-endian

16-bit word mix of both

3B 4A 1D 2C
2C 1D 4A 3B

PDP-11

Function and reference pointers


DOOM Shareware Startup v1.9
50
AMMO
100%
HEALTH
2 3 4
5 6 7
ARMS
0%
ARMOR
BULL 50 / 200
SHEL 0 / 50
ROKT 0 / 50
CELL 0 / 300

Statusbar value update

```
// health percentage
STlib_initPercent(&w_health,
    ST_HEALTHX,
    ST_HEALTHY,
    tallnum,
    &plyr->health,
    &st_statusbaron,
    tallpercent);
```

Statusbar value update

```
private class PercentageWidget
{
    public PercentageWidget(ref int number)
    {
        Number = number;
    }

    public int Number { get; set; }
}
```

Statusbar value update

```
private class PercentageWidget2
{
    public PercentageWidget2(Reference<int> number)
    {
        Number = number;
    }

    public Reference<int> Number { get; }

}

private class Reference<T> where T : struct
{
    public T Value { get; set; }
}
```

Statusbar value update

```
private class PercentWidget
{
    public PercentWidget(Func<int> numFunc)
    {
        NumFunc = numFunc;
    }

    public Func<int> NumFunc { get; set; }
}
```

Statusbar value update

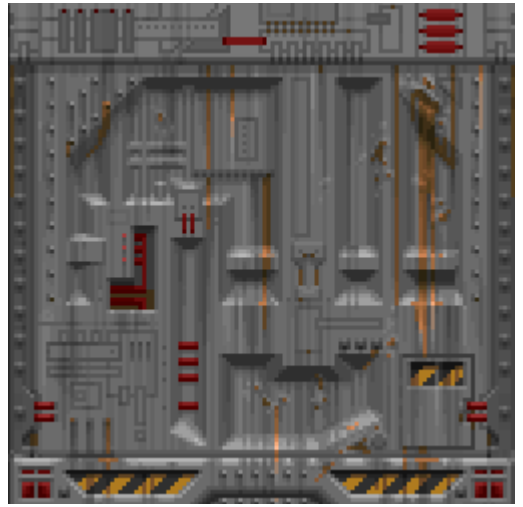
```
// health percentage
_healthWidget = new PercentWidget(
    ST_HEALTHX,
    ST_HEALTHY,
    _tallNum!,
    () => _player.Health,
    () => _statusBarOn,
    _tallPercent.Value
);
```

Statusbar value update

```
// health percentage
STlib_initPercent(&w_health,
    ST_HEALTHX,
    ST_HEALTHY,
    tallnum,
    &plyr->health,
    &st_statusbaron,
    tallpercent);
```

Thinkers and map objects

- Thinkers are things that react in a level:
- Doors can be opened or closed
- Ceilings can start coming down on you
- Floors can become staircases



Thinkers and map objects

- Map objects are special thinkers:
- The player is a map object, so are all of the enemies you're facing in DOOM
- Map objects get the `P_MapObjectThinker` method assigned to them

Thinkers and map objects

```
// Doubly linked list of actors.
typedef struct thinker_s
{
    struct thinker_s* prev;
    struct thinker_s* next;
    actionf_t         function;
} thinker_t;
```

```
typedef union
{
  actionf_p1 acp1;
  actionf_v  acv;
  actionf_p2 acp2;
} actionf_t;

typedef void (*actionf_v)();
typedef void (*actionf_p1)(void*);
typedef void (*actionf_p2)(void*,
void*);
```

Thinkers and map objects

```
// Doubly linked list of actors.
typedef struct thinker_s
{
    struct thinker_s* prev;
    struct thinker_s* next;
    actionf_t         function;
} thinker_t;
```

```
typedef union
{
  actionf_p1 acp1;
  actionf_v  acv;
  actionf_p2 acp2;
} actionf_t;

typedef void (*actionf_v)();
typedef void (*actionf_p1)(void*);
typedef void (*actionf_p2)(void*,
void*);
```

Thinkers and map objects

```
// Doubly linked list of actors.
typedef struct thinker_s
{
    struct thinker_s* prev;
    struct thinker_s* next;
    actionf_t         function;
} thinker_t;
```

```
typedef union
{
  actionf_p1 acp1;
  actionf_v  acv;
  actionf_p2 acp2;
} actionf_t;

typedef void (*actionf_v)();
typedef void (*actionf_p1)(void*);
typedef void (*actionf_p2)(void*,
void*);
```

Thinkers and map objects

```
// Doubly linked list of actors.
typedef struct thinker_s
{
    struct thinker_s* prev;
    struct thinker_s* next;
    actionf_t         function;
} thinker_t;
```

```
typedef union
{
  actionf_p1 acp1;
  actionf_v  acv;
  actionf_p2 acp2;
} actionf_t;

typedef void (*actionf_v)();
typedef void (*actionf_p1)(void*);
typedef void (*actionf_p2)(void*,
void*);
```

Thinkers and map objects

```
public abstract class Thinker
{
    public Action<ActionParams>? Action { get; set; }
}

public record ActionParams(MapObject? MapObject = null, Player? Player = null,
PlayerSprite? PlayerSprite = null, Thinker? Thinker = null);
```

Thinkers and map objects

```
public abstract class Thinker
{
    public Action<ActionParams>? Action { get; set; }
}

public record ActionParams(MapObject? MapObject = null, Player? Player = null,
PlayerSprite? PlayerSprite = null, Thinker? Thinker = null);
```

Thinkers and map objects

```
public abstract class Thinker
{
    public Action<ActionParams>? Action { get; set; }
}

public record ActionParams(MapObject? MapObject = null, Player? Player = null,
PlayerSprite? PlayerSprite = null, Thinker? Thinker = null);
```

Thinkers and map objects

```
public abstract class Thinker
{
    public Action<ActionParams>? Action { get; set; }
}

public record ActionParams(MapObject? MapObject = null, Player? Player = null,
PlayerSprite? PlayerSprite = null, Thinker? Thinker = null);
```

Thinkers and map objects

```
public abstract class Thinker
{
    public Action<ActionParams>? Action { get; set; }
}

public record ActionParams(MapObject? MapObject = null, Player? Player = null,
PlayerSprite? PlayerSprite = null, Thinker? Thinker = null);
```

Thinkers and map objects

```
public abstract class Thinker
{
    public Action<ActionParams>? Action { get; set; }
}

public record ActionParams(MapObject? MapObject = null, Player? Player = null,
PlayerSprite? PlayerSprite = null, Thinker? Thinker = null);
```

Thinkers and map objects

```
public abstract class Thinker
{
    public Action<ActionParams>? Action { get; set; }
}

public record ActionParams(MapObject? MapObject = null, Player? Player = null,
PlayerSprite? PlayerSprite = null, Thinker? Thinker = null);
```

Thinkers and map objects

```
public static void VerticalDoor(ActionParams actionParams)
{
    if (actionParams.Thinker is not Door door)
    {
        return;
    }

    switch (door.Direction)
    {
        case 0:
            // waiting
            if (--door.TopCountDown == 0)
            // ...
```

Thinkers and map objects

```
public static void VerticalDoor(ActionParams actionParams)
{
    if (actionParams.Thinker is not Door door)
    {
        return;
    }

    switch (door.Direction)
    {
        case 0:
            // waiting
            if (--door.TopCountDown == 0)
            // ...
```

Any questions?

Xebia

Thanks for listening!