Game Engine Architecture

Chapter 6

Engine Support Systems

Overview

- Subsystem Start-up and shut down
- Memory Management
- Containers
- Strings
- Engine Configuration

Start-up and shut-down

- Game engines are complex beasts
- When it first starts up a number of subsystems must be configured and initialized
- Order is relevant because of subsystem dependencies
- Typically started in one direction and shutdown in the opposite

Singleton

- A common design pattern for implementing major subsystems such as the ones that make up a game engine is to define a *singleton class* (often called a *manager*) for each subsystem

```
class RenderManager{
        public:
        RenderManager (){
                //start up the manager
        }
        ~RenderManager (){
                //shut down the manager
        }
};
//singleton instance
static RenderManager gRenderManager
```

- If C++ gave us more control over the order in which global and static class instances were constructed and destroyed, we could define our singleton instances as globals, without the need for dynamic memory allocation.

Controlling the singleton

- One trick you can use is that a static variable declared within a method is not initialized at startup

```
class RenderManager{
    public:
    static RenderManager& get(){
        static RenderManager sSingleton;
        return sSingleton;
    }
    RenderManager (){
        VideoManager::get();
        TextureManager::get();
    }
    ~RenderManager (){
        //shut down the manager
    }
};
```

- This works pretty well, but you cannot control the shutdown at all

Do it directly

- Create startup and shutdown methods in all of the managers
  - It's simple
  - It's understandable
  - Easy to debug and maintain
- You could also use singletons where the main creates the objects using new

A Simple Approach That Works

```
class RenderManager
{
public:
RenderManager()
{
// do nothing
}
~RenderManager()
{
// do nothing
}
void startUp()
{
// start up the manager...
}
void shutDown()
{
// shut down the manager...
}
// ...
};
```

- To Define explicit start-up and shut-down functions for each singleton manager class
- These functions take the place of the constructor and destructor, and in fact we should arrange for the constructor and destructor to do *absolutely nothing*.
- That way, the start-up and shut-down functions can be explicitly called *in the required order* from within main()

brute-force approach

```
class PhysicsManager { /*
similar... */ };
class AnimationManager { /*
similar... */ };
class MemoryManager { /*
similar... */ };
class FileSystemManager { /*
similar... */ };
// ...
RenderManager gRenderManager;
PhysicsManager gPhysicsManager;
AnimationManager
gAnimationManager;
TextureManager gTextureManager;
VideoManager gVideoManager;
MemoryManager gMemoryManager;
FileSystemManager
gFileSystemManager;
```

```
int main(int argc, const char* argv)
{
// Start up engine systems in the correct order.
gMemoryManager.startUp();
gFileSystemManager.startUp();
gVideoManager.startUp();
gTextureManager.startUp();
gRenderManager.startUp();
gAnimationManager.startUp();
gPhysicsManager.startUp();
// ...
// Run the game.
gSimulationManager.run();
// Shut everything down, in reverse order.
// ...
gPhysicsManager.shutDown();
gAnimationManager.shutDown();
gRenderManager.shutDown();
gFileSystemManager.shutDown();
gMemoryManager.shutDown();
return 0;
}
```

brute-force approach

- Pros
  - It's simple and easy to implement.
  - It's explicit. You can see and understand the start-up order immediately by just looking at the code.
  - It's easy to debug and maintain. If something isn't starting early enough or is starting too early, you can just move one line of code.
- Cons
  - One minor disadvantage to the brute-force manual start-up and shut-down method is that you might accidentally shut things down in an order that isn't strictly the reverse of the start-up order.

Memory Management

- The performance of your game engine is associated not only with your algorithms, but the way you use memory
  - malloc() and **new** are slow
  - Access pattern and memory fragmentation seriously impact caching performance

Dynamic Allocation

- Heap allocation is slow because it has to be general
  - Handles requests from 1 byte to 1GB
  - Creates a ton of management overhead
- Also slow because of a context switch
  - First switches from user to kernel mode
  - Then has to switch back
- Cannot completely avoid dynamic allocation, but can create custom allocators to avoid it

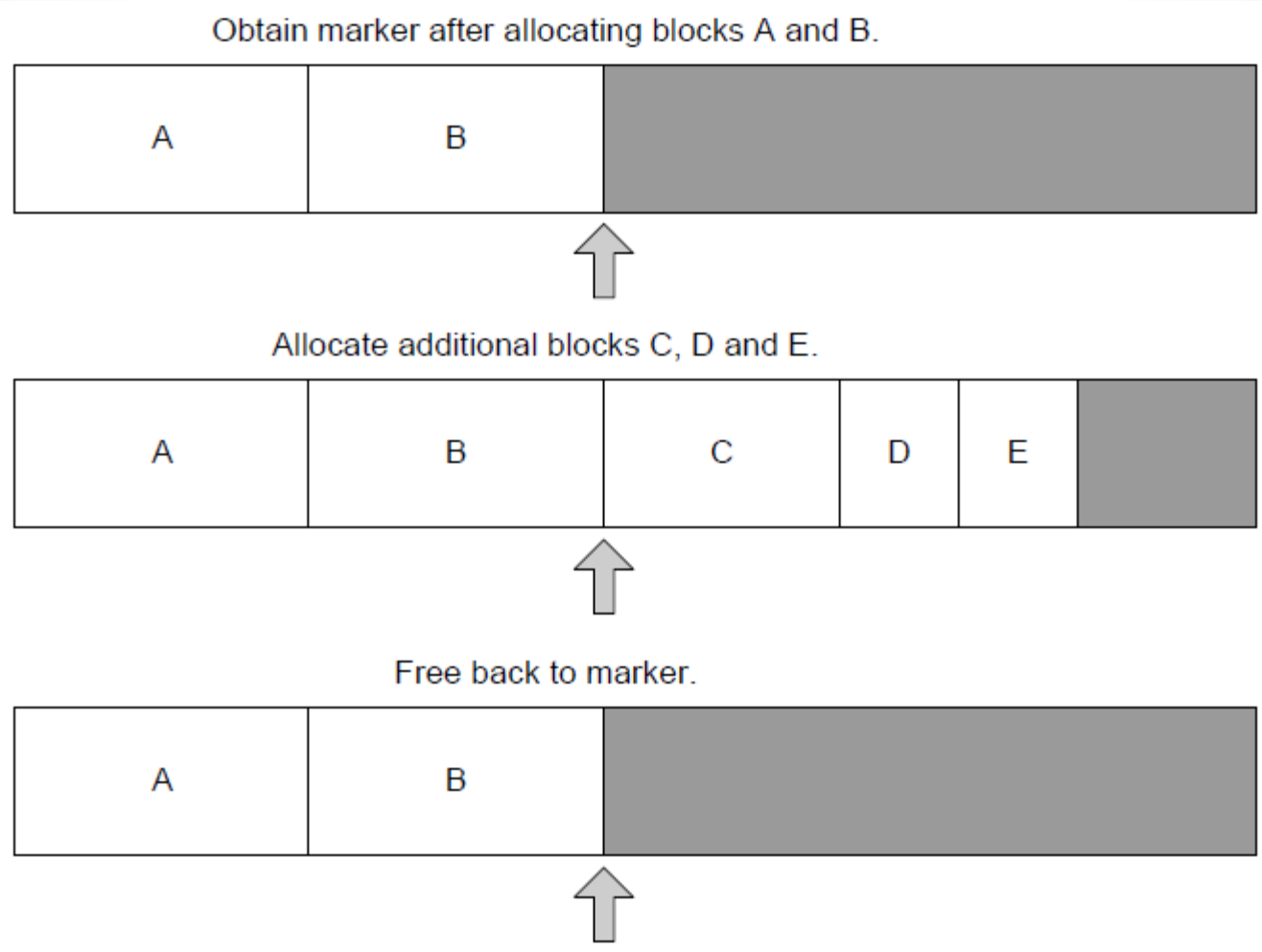
Stack-based Allocator

- The easiest way is to use a pre-allocated memory block and use it as a stack
- When a level is loaded, add it to the stack, when it is finished, move the stack pointer back
- Order is important because you can overwrite a current used memory location
- Often done using rollback markers

Stack-based Allocator

```
class StackAllocator
{
public:
// Stack marker: Represents the current top of the stack. You can only roll back to a marker, not to
// arbitrary locations within the stack.

typedef U32 Marker;

// Constructs a stack allocator with the given total size.

explicit StackAllocator(U32 stackSize_bytes);

// Allocates a new block of the given size from stack top.
void* alloc(U32 size_bytes);

// Returns a marker to the current stack top.
Marker getMarker();

// Rolls the stack back to a previous marker.
void freeToMarker(Marker marker);

// Clears the entire stack (rolls the stack back to zero).
void clear();

private:
// ...
};
```

Stacks

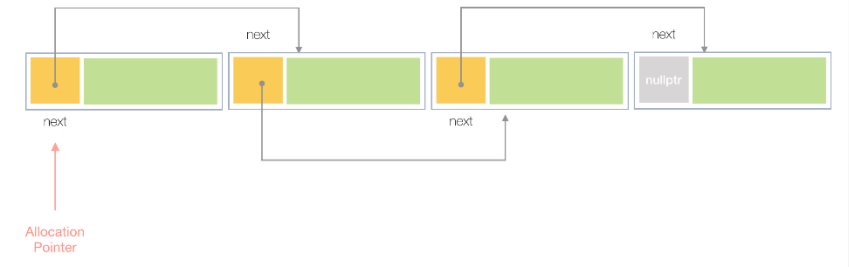

Obtain marker after allocating blocks A and B.
A
B
Allocate additional blocks C, D and E.
A
B
C
D
E
Free back to marker.
A
B

Double ended

- Another method uses a double ended stack
  - Useful for having big allocations on one side and small temporary on the other
- A single memory block can actually contain two stack allocators
- one that allocates up from the bottom of the block and one that allocates down from the top of the block.
- both stacks may use roughly the same amount of memory and meet in the middle of the block

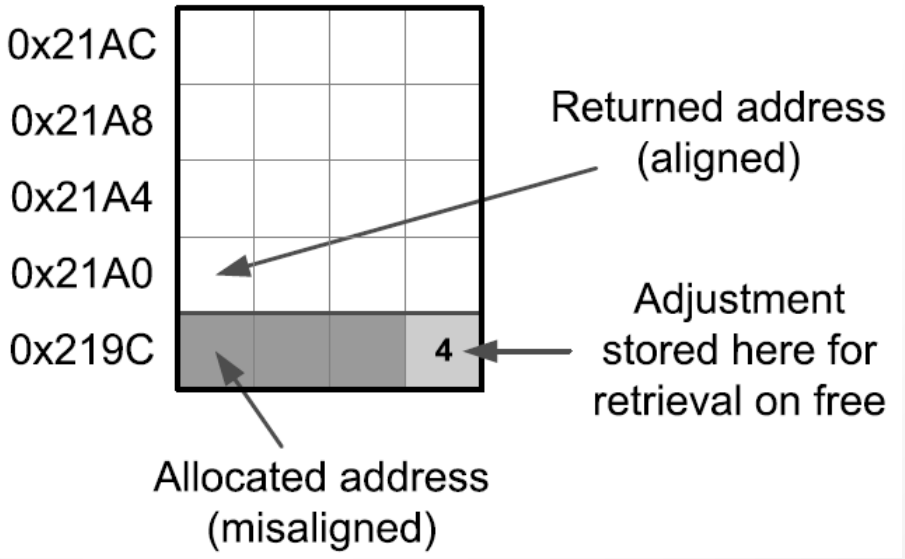
Pool Allocator

- This technique works by allocating a large number of fixed sized memory blocks
  - 1,000 4X4 matrices
  - For example, a pool of 4X4 matrices would be an exact multiple of 64 bytes—that's 16 elements per matrix times four bytes per element
- When you need a new matrix, you get it from the pool
- Return it to the pool when are done
- The pool can be managed by a linked list
- Each element within the pool is added to a linked
- list of free elements;
- when the pool is first initialized, the free list contains all of the elements.



Aligned Allocation

- One problem with pooled memory is that every variable and object has an alignment requirement
- The memory allocator must be able to return aligned memory otherwise you have serious trouble
- Not a serious problem as an aligned allocator is easy to write

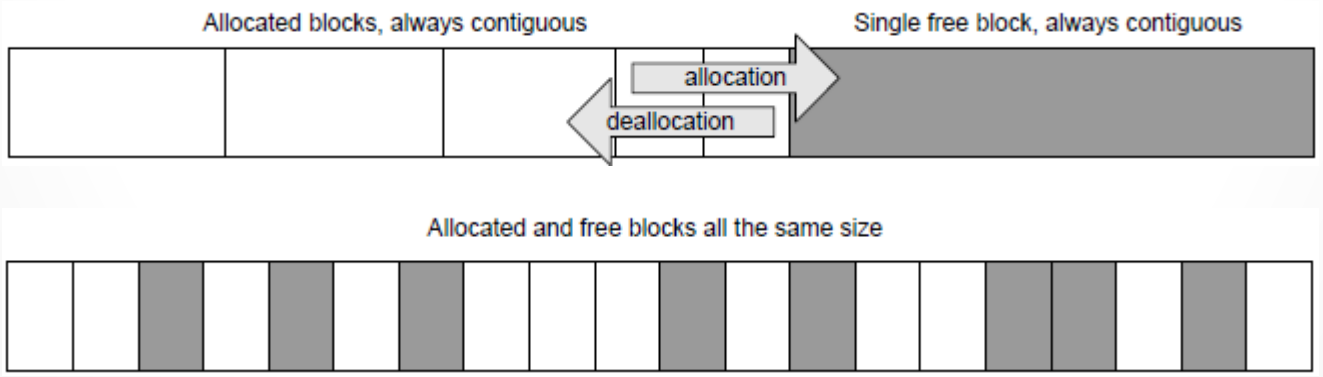


Single or double frame

- You often need to allocate memory every frame One way to do this is to use a single frame buffer
- Allocate the memory once and free it only when the rendering is complete
  - Very fast, you have to be careful
- A double frame buffer might be better in a multi-core setup
  - Allocate memory at frame i for use in i+1

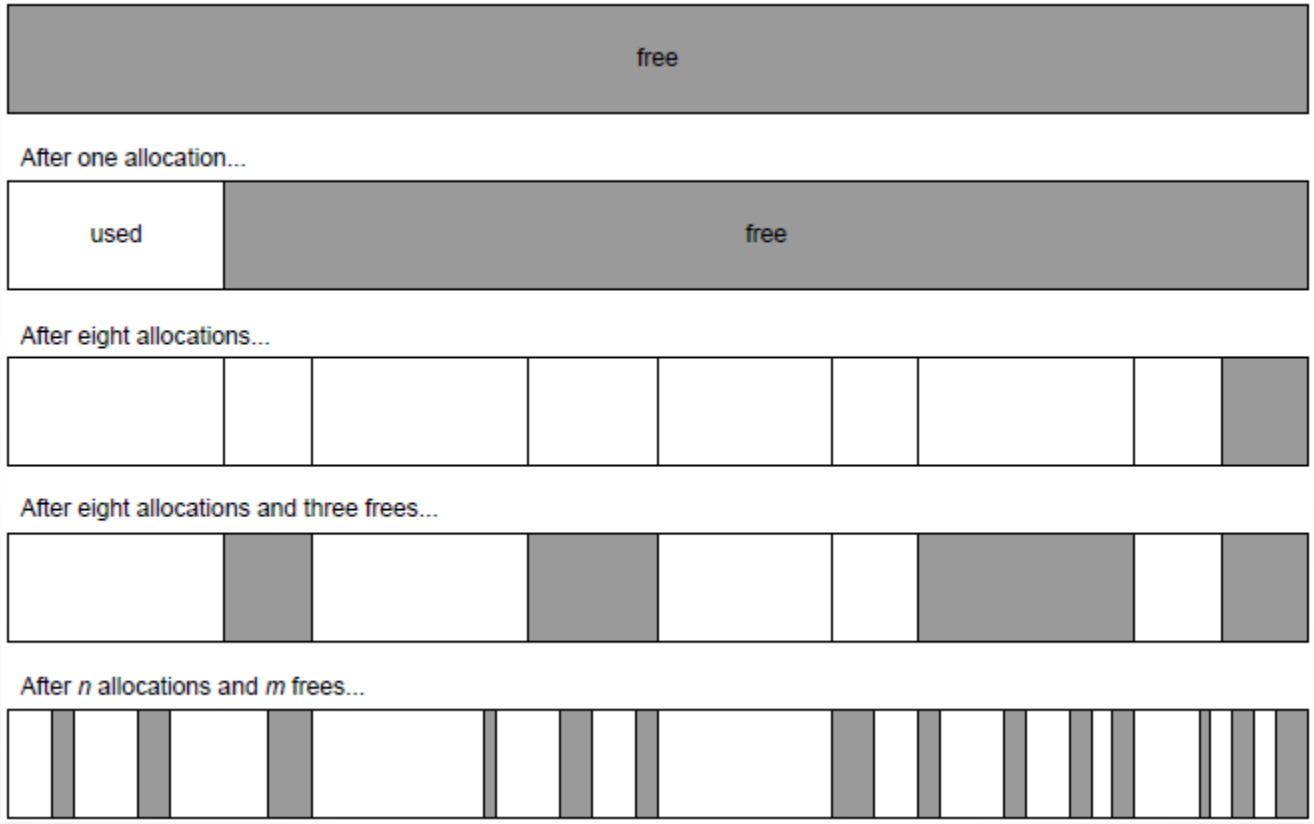
Fragmentation

- Doing dynamic allocation can create memory fragments
  - This slows down memory copies
  - Can prevent allocation when a contiguous block is not available
- Pooled and stack allocators avoid this problem

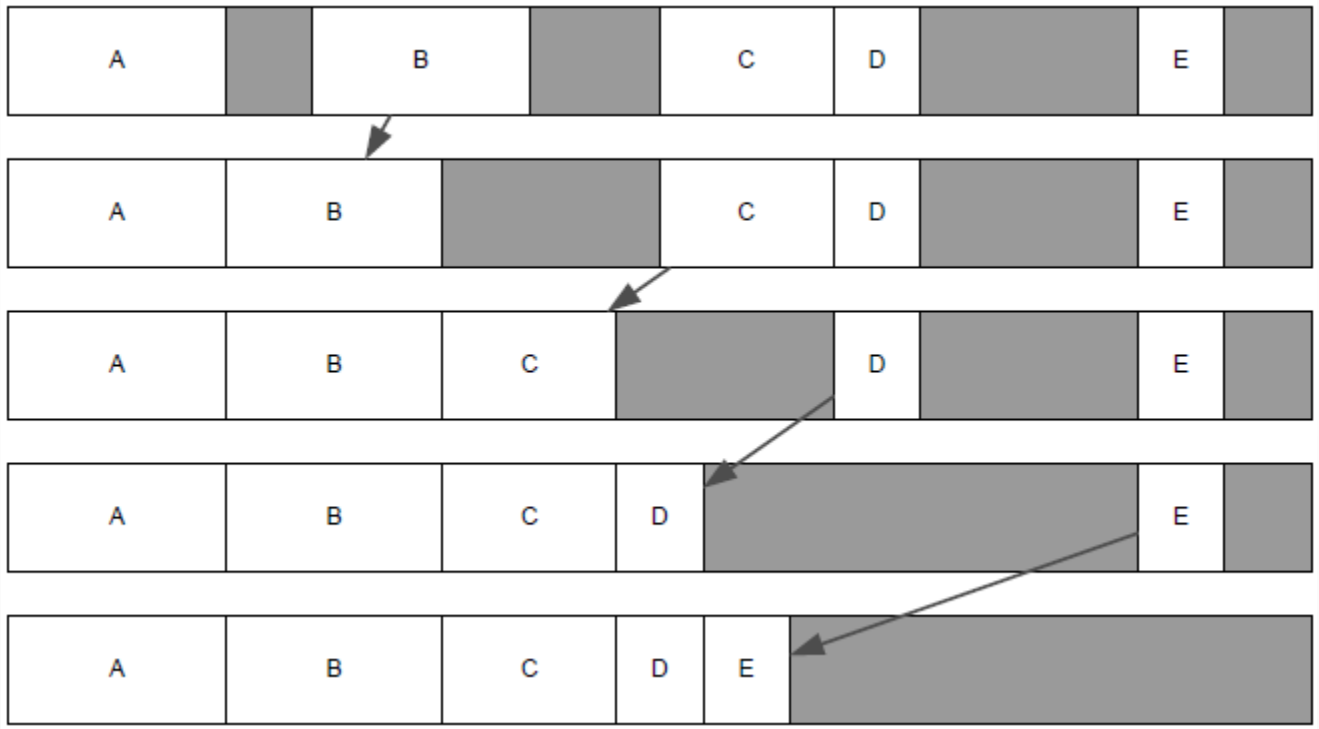


Fragmentation

- If you need random allocation/deallocation you may require a defragmentation routine



Shift



Defrag

- The defrag operation can occur over several frames
  - Just move one thing at a time
- Moving memory is tricky though
  - All pointers to it need to be updated
- Handles are frequently used
  - Handles point to immutable memory that contains the current pointer to the object (yes, a pointer to a pointer)

Cache

- By now, you should understand how cache works
- To avoid cache misses on data, we try to keep data chunks small, contiguous in memory, and access them sequentially
- Instructions are also held in cache

I-Cache

- The compiler and linker handle most of the details about how code is represented in memory
- We can help it because it follows certain rules
  1. Machine code from a function is contiguous in memory
  2. Functions are stored in their original order
  3. Functions in a single file are almost always contiguous

Taking Advantage

- Keep high-performance code small
- Avoid making function calls in a performance critical section
- If you have to call a function, put it as close as possible to the caller and never in another file
- Don't overuse inline functions, they can bloat the code

Containers

- Games developers use many different data structures
  - Array
  - Dynamic Array
  - Linked List
  - Queue
  - Deque
  - Tree
  - Binary Search Tree
  - Binary heap
  - Priority Queue
  - Dictionary
  - Set
  - Graph
  - Directed acyclic graph
- Many of these can be found in STL (Standard Template Library)

Container Operations

- Insert
- Remove
- Sequential access (iteration)
- Random access
- Find
- Sort

- Remember that different containers have different costs for these various operations

Iterators

- STL has an iterator class that works much like the one in Java
- Allows you to maintain encapsulation in the container object
- Easier to use than pointer manipulation

```
void processList (std::list<int>& container)
{
        std::list<int>::iterator pBegin = container.begin();
        std::list<int>::iterator pEnd = container.end();
        std::list<int>::iterator p;
        for (p = pBegin; p !=pEnd; p++)
        {
                int element = *p;
        }
}
```

Building Custom Containers

Many reasons to do so

1. *Total control* – you dictate the algorithms, memory use, etc.
2. *Opportunities to optimize* – optimize based on a specific hardware platform
3. *Customizability* – can add custom features specific to your purpose
4. *Elimination of external dependencies* – no licensing fees, you can fix it yourself
5. *Control of concurrent data structures* – you have total control over concurrent access

Standard Template Library

- Benefits
  - Rich set of features
  - Robust implementation on a wide variety of platforms
  - Comes standard with most C++ compilers
- Drawbacks
  - Steep learning curve
  - Often slower than a custom crafted data structure
  - Eats up a lot of memory
  - Does a lot of dynamic memory allocation
  - Performance varies based on the compiler

Rules for using STL

- Be aware of the performance and memory characteristics
- Avoid heavyweight STL classes in critical code sections
- Don't use it when memory is at a premium
- Use STLPort if you plan to create multiplatform games

Boost

- Aim was to build libraries that extend and work with STL
- Benefits
  - Provides things not available in STL
  - Provides some workarounds to problems with STL
  - Handles complex problems like smart pointers
  - Documentation is really good
- Drawbacks
  - Most core classes are templates – has large .lib files
  - No guarantees – if you find a bug, it's your issue
  - Boost license – not very restrictive
- https://www.boost.org/

Loki

- Written by Andrei Alexandrescu
- Very powerful, but hard to understand
- Less portable because it uses sophisticated compiler tricks
- Look for the book *Modern C++ Design* by Alexandrescu

Strings

- Strings are extremely important in games
- They are far from simple to manage
  - What if they need to be resized?
  - How do you deal with *localization* issues?
    - Different character sets
    - Different lengths for translations
    - Different display layouts
- Checking for equality is an O(n) operation

String Classes

- They come with an overhead
  - Copy constructors on a function call
  - Dynamic memory allocation
- Probably should be avoided by using fixed sized wchar_t arrays
- Path classes might be the exception
  - Often include more information than a String

Unique IDs

- Objects within the game need a way to be identified
- Strings seem like a natural choice
  - Comparison costs are not good
- GUIDs (numbers) are a lot faster to compare
  - Harder to remember

Hashed String IDs

- We can use hash functions to map our strings to numbers
  - Best of both worlds
- Collisions are possible
  - A good hash function eliminates this concern
  - Using an uint32 for the values gives over 4 million possible values
- These are sometimes referred to as a *string id*

Implementation ideas

- Runtime hashing can be slow
  - Doing many of them can take a long time
- One way to avoid this is to offline process the source code
  - Look for occurrences of the function call and replace it with the hashed number
- Another way is to create a static variable to *intern* the string ids

Localization

- Best to plan for localization from day 1
- Important to understand that ASCII character codes don't support localization at all
- Retrain your brain to think in Unicode

Unicode

- Like ASCII, unicode assigns a unique *code point* to every character or glyph
- When storing characters we use a particular *encoding*
- The combination of encoding and code point yields a character or glyph
- Common encodings are
  - UTF-32
  - UTF-8
  - UTF-16

UTF-32

- The simplest encoding because all code points are stored in a 32-bit value
- Wasteful
  - Most western languages don't use high value code points (wastes 2 bytes per character)
  - The highest Unicode code point is 0x10FFFF (only 21 bits)
- Easy because we can figure out the length of a string by dividing number of bytes by 4

UTF-8

- Code points are stored in one-byte granularity, but some use two
- It's called *variable length encoding* or *multibyte character set* (MBCS)
- It's backwards compatible with ANSI encoding
  - Needs 7 bits to represent the 127 characters
- Multibyte characters have the first bit set to one
  - This indicates that there are two bytes in the code point

UTF-16

- A bit simpler than UTF-8, but more expensive
- Code points stored as one or two 16-bit values
- Also called Wide Character Set (WCS)
- UTF contains 17 *planes* that each contain 2^{16} code points
- First plane called basic multilingual plane (BMP)
  - Most characters are present in this plane
- The other planes are called supplementary planes
  - Requires two 16-bit values

UCS-2

- A subset of UTF-16 containing only the BMP
- Its main advantage is that it is fixed length
  - UTF-16 and UTF-8 are variable length
- Can be stored in little endian or big endian
  - Often stored with a Byte Order Marker (BOM)

char and wchar_t

- Standard C/C++ define two data types for characters
  - *char* is used to for legacy ANSI strings or for MBCS
  - *wchar_t* is used to represent any valid code point
    - Could be 8, 16, or 32 bits
- To write truly platform independent code you will need to define your own character data types

Unicode in Windows

- In Windows *wchar_t* is exclusively for UTF-16 encoding and *char* for ANSI encoding
- Windows API defines three sets of character/string functions

ANSI	WCS	MBCS
strcmp()	wcscmp()	_mbscmp()
strcpy()	wcscpy()	_mbscpy()
strlen()	wcslen()	_mbstrlen()

- There are also translation functions like wcstombs()

Unicode on Consoles

- Xbox360 uses WCS strings
- At Naughty Dog they only use char strings
  - Foreign languages are handled with UTF-8 encoding

Other Localization Concerns

- Need to translate more than strings
  - Audio clips
  - Textures – if they have English words on them
  - Symbols – may not mean what you think it means
  - Market specific game-rating issues – blood changes the teen-rating in Japan
- Localization database
  - Need a way to convert string ids to human readable strings
  - Form is up to you – Varies from CSV to full databases

Id	English	French
p1score	“Player 1 Score”	“Grade Joueur 1”
p2score	“Player 2 Score”	“Grade Joueur 2”
p1wins	“Player one wins!”	“Joueur un gagne!”
p2wins	“Player two wins!”	“Joueur deux gagne!”

Final notes

- Establish a set of functions early to handle localization
- Force developers to use those functions instead of using string literals in the code
- Create a configuration system to allow the language to be set

Engine Configuration

- Most engines require the ability to save and load configuration files
- Many ways to do this
  - Text files
  - Compressed binary files
  - Windows registry
  - Command line options
  - Environmental variables
  - Online user profiles

Game vs User options

- Be careful to separate the game settings from those of a particular user
- On windows machines you can use ApplicationData directory by creating your own folder
- You can also use the special key HKEY_CURRENT_USER in the registry to store settings information

Examples

- Quake uses Cvars – named values with a set of flags
  - These are stored in a linked list and the values are retrieved by name
  - The flags indicate if the value should persist – written to file
- Ogre3D
  - Uses text files in the Windows INI format
- Uncharted
  - Several mechanisms including Scheme