Sekai Engine

Developer reference

2007-2009

Table of contents

[1 Introduction 4](#_Toc236736080)

[1.1 Architecture overview 5](#_Toc236736081)

[1.2 Main components 6](#_Toc236736082)

[2 Core layer 7](#_Toc236736083)

[2.1 Core 7](#_Toc236736084)

[2.1.1 Extension points 7](#_Toc236736085)

[2.1.2 Plug-in graph 11](#_Toc236736086)

[2.1.3 Global environment 12](#_Toc236736087)

[2.2 Interfaces and object exporting 13](#_Toc236736088)

[2.2.1 SCOM Library 13](#_Toc236736089)

[2.2.2 Using The library 14](#_Toc236736090)

[2.2.3 Intellectual types 18](#_Toc236736091)

[2.3 Reflection 20](#_Toc236736092)

[2.3.1 Idea and purpose of reflection 20](#_Toc236736093)

[2.3.2 Library Structure 21](#_Toc236736094)

[2.4 Script engine 25](#_Toc236736095)

[2.4.1 Advantages of scripting 25](#_Toc236736096)

[2.4.2 Script integration 25](#_Toc236736097)

[2.4.3 Scripting language 25](#_Toc236736098)

[2.4.4 Python binding implementation 26](#_Toc236736099)

[3 Service libraries overview 29](#_Toc236736100)

[3.1 Logging library 29](#_Toc236736101)

[3.2 File system library 29](#_Toc236736102)

[3.3 Math library 30](#_Toc236736103)

[3.4 Time library 31](#_Toc236736104)

[3.4.1 Timers 31](#_Toc236736105)

[3.4.2 Interpolation framework 31](#_Toc236736106)

[4 Engine layer 34](#_Toc236736107)

[4.1 Graphics engine 34](#_Toc236736108)

[4.1.1 Design approach 34](#_Toc236736109)

[4.1.2 Renderer interface 35](#_Toc236736110)

[4.1.2 Resource managers 36](#_Toc236736111)

[4.1.3 Resources 37](#_Toc236736112)

[4.1.4 Rendering process 39](#_Toc236736113)

[4.2 Input library 42](#_Toc236736114)

[4.3 Main engine module 43](#_Toc236736115)

[4.3.1 Module goals 43](#_Toc236736116)

[4.3.2 Engine object 43](#_Toc236736117)

[4.3.3 Entity structure 44](#_Toc236736118)

[4.3.4 Built-in components 45](#_Toc236736119)

[4.3.5 Creating own components 48](#_Toc236736120)

[4.4 Logic Graph 49](#_Toc236736121)

[4.5 Shading Graph 50](#_Toc236736122)

[5 User Interface Layer 51](#_Toc236736123)

[5.1 Vector Graphics Library 51](#_Toc236736124)

[5.2 Font engine 54](#_Toc236736125)

[Appendix A: Setting up the environment 55](#_Toc236736126)

[External dependencies 55](#_Toc236736127)

[Building STLPort and Boost libraries 56](#_Toc236736128)

[Appendix B: Thoughts for the future 58](#_Toc236736129)

[Greedy Material System 58](#_Toc236736130)

# 1 Introduction

## 1.1 Architecture overview

Main goal in the engine design was maximum scalability and flexibility. That’s why the plug-in based architecture approach was chosen. See Extension Points section for more information on plug-in based architecture concept.

Common approach to plug-in architectures is to allow extending some specific places in the system (typically by implementing specific interface and placing library at some defined folder where the system will be looking for extensions). This is a widely used approach, however it has several flaws.

All implementations of the extension points (places in the system that can be extended via plug-ins) have its own rules of plug-in look-up and creation logic. This means that developer, who already wrote an extension to one part of the system, cannot write a plug-in to another part without consulting about how the provided implementation will be searched for, or what functionality it should export to be constructed. This can lead to heterogeneous, inconsistent system structure, where every subsystem define own set of interaction rules.

Another example is an overuse of access to implementation details. Consider the case when some part of the system should be extendable by plug-ins, and, at the same time, some default or predefined functionality should be added too. In this situation most of developers will hardcode the predefined part, what “splits” the implementation in to half, making it hard to support. “Fair play” principle states that developer should use he’s own extension point to provide the extension he develops. But again, without the single extension mechanism it is almost impossible.

In plug-in based system[[1]](#footnote-1) any set of functionality is a plug-in. This means that throughout all system exist the single well-defined mechanism to provide an extension point and to extend them. In fact the border between the extension and extension provider is blurred, because any plug-in can extend and provide extension point at the same time.

This approach leads to consistent and uniform system design. Sekai engine is built of multiple modules that bind together in the runtime by Core. Core is the only static component of the system. The module binding is driven by plug-in description files, which will be covered in next sections.

Because plug-ins are often compiled separately from the Core, or other plug-ins, it is necessary for all of them to hide their implementation details. To achieve binary-level encapsulation[[2]](#footnote-2) modules reference each-other only using interfaces, so any module can be changed and recompiled without the need of recompilation of all modules that depend on it.

## 1.2 Main components

* Core - binds all plug-ins together, holds the relation graph of plug-ins;
* Core.SCOM - lightweight implementation of COM, supports interface querying, factories and module maps;
* Core.Logging - provides logging facilities, also supports exception handling with mini-dump and call-stack generation;
* Core.FileSystem - provides platform abstraction of file system, also supports file filter mechanism, archives, and asynchronous loading;
* Core.Reflection – provides means for non-intrusive reflection (introspection) of C++ types, function invocation and events are here too;
* Core.Script – implementation of own-developed method of reflection-based script binding generation. Now implemented version for Python 3.1;
* Core.Math - vector math library, provides optimized math routines for SSE and SPU (Play Station 3 cell processor);
* Core.Time – provides time measuring facilities and interpolation framework;
* Core.Utils - utility classes, functors etc.;
* Engine - frame loop, scene management, game objects, space partition and many more;
* Engine.Graphics - provides rendering API abstractions, material system etc.;
* Engine.Graphics.VG - low-level library for vector graphics rendering, in future will be used in GUI subsystem;
* Engine.Input - abstract interface to input devices;

Below is a high-level representation of modules, modules colored gray are planned but not implemented yet:



Figure 1.1 – Engine component diagram

# 2 Core layer

Core layer is the lowest level of module hierarchy. The main component of this layer is the Core itself. This layer also contains some low-level libraries all of which will be covered in this chapter.

## 2.1 Core

Library features:

* Plug-in based system composition approach
* Uniform way to add functionality
* Strict separation of modules and lowest possible coupling
* "Lazy” loading of plug-ins
* Easy access to major system components in layer-preserving way

As was said before, Core is the only static component of the system. Its task is to load all modules and arrange their interactions.

At startup, Core generates so called Plug-in Graph. Plug-in Graph defines the dependencies between modules, extension points they provide, and points they extend. This information is formed using the meta-info inside the plug-in description XML files, without loading of actual library.

### 2.1.1 Extension points

Extension point (EP) can be described as a contract between the plug-in that provides the point, and those who will extend it. Plug-in that define the extension point defines the interface which all extenders should implement.

Extension point provider may also define a specific set of parameters, which extenders should specify in their description files. Those parameters are mainly used to increase the “laziness” of loading (loading of actual library is delayed as much as possible). Extenders, in their turn, should define all the parameters and the class, which implements the EP’s interface.

Because extension parameters should be accessible even without loading of the library, in Sekai they are defined in the plug-in description files. These files are a simple XML, which define all necessary meta-information about the extension.

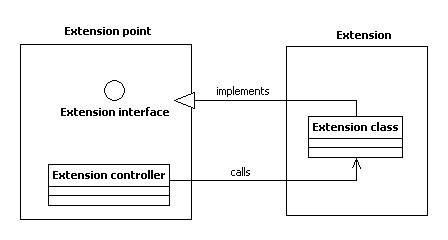


Figure 2.1 – Extension point representation

Extension point–extender architecture was in fact so versatile, that it was used in several more projects (like compilers, IDE, and desktop applications) without any changes in the core.

To clarify the module interaction let’s use the example from the IDE similar to Visual Studio. Assume that module that holds the project structure want to extend the toolbar, to add the “Compile” button to it:

1. Main window defines extension point MainWindow::ToolBar;
2. For lazy loading, point defines such parameters that should be specified in plug-in definition file by extenders: icon, tooltip text, group;
3. All extenders should implement IAction interface;
4. Project module author want to add compilation button to toolbar;
5. He implements IAction interface in one of he’s classes and adds this class to Module Map;
6. He describes the extension in plug-in description file, along with exported class and parameters;
7. At system startup Core gathers data from all description files, in particular, adds the connection between EP MainWindow::ToolBar and its extension in project plug-in;
8. At main window loading time, toolbar manager combines toolbar from all extenders using only the meta-data from plug-in graph;
9. Only when the button is actually pressed, and if project module was not loaded before, Core initiates the module loading. Toolbar managers requests the instance of implementation class, associated with the extension, and checks that it implements desired IAction interface;
10. Now, having the right implementation class, Toolbar manager notifies it about the action using interface methods.



Figure 2.2 – Extension point example

This process may look complex and slow, but it takes place only at startup and first usage of some module. Further all data is cached, so it will not affect the performance in computationally-intensive parts of system. Small drawback in performance, compared to static binding, is nothing for such a huge bonus in extensibility.

Here’s how description file of main window plug-in from previous example will look like:

<?xml version="1.0" encoding="utf-8"?>

<plugin

name="MainWindow"

brief="Main window plug-in"

version="1"

author="anonymus">

<provides point="ToolBar" interface="IAction" iid="…">

<id>unique action id </id>

<name>action name</name>

<group>group name, for separated groups</group>

<icon>relative icon path</icon>

</provides>

...

</plugin>

And here’s the description file of project plug-in, that will extend the ToolBar extension point:

<?xml version="1.0" encoding="utf-8"?>

<plugin

name="Project"

brief="Project graph plug-in"

version="1"

author="anonymus">

<exports>

<class name="CProjectGraph" classid="…"/>

</exports>

<extends point="MainWindow::ToolBar" classid="CProjectGraph">

<id>Project.Compile</id>

<name>Compile</name>

<group>Project</group>

<icon>Icons/Compile.ico</icon>

<extends/>

...

</plugin>

All entries in XML are pretty self-explanatory, below is the common format of plug-in description files:

<?xml version="1.0" encoding="utf-8"?>

<plugin

name="Sample.plugin"

brief="Sample plug-in"

version="1"

author="Sample writing guy">

<!--Required plug-ins to run this one, version parameter is optional-->

<prerequisites>

<uses plugin="core.filesystem" version="1"/>

</prerequisites>

<!--Exports such classes-->

<exports>

<class name="Class1" classid="e208e49e-0b00-45ff-bc17-a23500f730e2"/>

<class name="Class2" classid="3f7943ba-d059-4b57-88dd-cdb8e5790fff"/>

</exports>

<!--Which EPs this plug-in extends-->

<!--'class' property can be omitted-->

<extends point="Someplugin::someextensionpoint" class="Class1">

<!--Here you can specify the params for the extendee, but ONLY in the form of name-value-->

<param1>value</param1>

<param2>value</param2>

<!--Values with names of exported classes will be replaced by classID strings from the export section-->

<impl>Class2</impl>

</extends>

<!--Defines new extension point-->

<!--interface name is ignored while parsing, it is just for convinience and static analysis-->

<provides point="testpoint\_listeners" interface="ITestPoinListener" iid="3f8143ba-a069-4b57-9a47-cdb8e5790afd">

<!--Here comes params this point expects-->

<!--Params can be only defined as key-value pairs-->

<!--They are ignored in parsing time-->

<listener>classname</listener>

</provides>

</plugin>

### 2.1.2 Plug-in graph

Most of the plug-in graph components already been covered in previous chapters, so here we will just use the collaboration diagram to clarify the roles of each object.



Figure 2.3 – Collaboration diagram of plug-in graph components

All interactions between plug-ins start with IPluginManager interface which can be accessed from any plug-in through global environment.

### 2.1.3 Global environment

If your plug-in exports class with CLSID\_Plugin ID, core will automatically create its implementation and call initialization method, which accepts global environment pointer. Common implementation of IPlugin will save this pointer to global gEnv variable.

Global environment ease the access to main subsystems, such as core, plug-in manager and logger, freeing you from multiple slow queries to the core. There is also a way to add more info to global environment. You can subclass GlobalEnvironment and then register new global environment in the core. After this, all newly created plug-ins will receive pointer to new, extended, environment. This allows adding subsystems to the environment in layer-preserving way.

More info you can get in API reference on GlobalEnvironment structure.

## 2.2 Interfaces and object exporting

Library features:

* Uniform way to export functionality from module
* Binary-level encapsulation
* Interface similar to COM/ATL
* Customizable (user can specify own factories, allocators, and thread models)
* GUID-based interface identification
* Fast type down casting

Sekai engine widely use interface-implementation paradigm for module interactions. Because Core and other modules are loosely coupled, it is important to add a compilation firewall, so that all modules can be individually recompiled without breaking each other. Independence of one module from implementation details of another is often called binary-level encapsulation.

### 2.2.1 SCOM Library

One of the bright examples of interface-implementation paradigm is COM (Component Object Model) from Microsoft. Pure COM is complex and hard to use, so in most cases it is used in combination with ATL (Active Template Library). COM suits well for developing of large systems, but it is heavyweight and not portable. So, Sekai uses its own COM-like library.



Figure 2.4 – SCOM library class diagram

Main components of the library:

* IUnknown interface – base for all interfaces in the system, provides reference counting and RTTI facilities;
* Template class ComRootObject – all implementation classes should inherit this class to get the default implementation of IUnknown methods. With template parameter you can specify the threading strategy of the reference counting;
* Interface map – table version of QueryInterface method, this table associates different interface IDs with the offset of their implementation from the beginning of the class;
* Module map – here should be defined all class IDs that can be created by module user. Objects are created through factories, this allows to apply different life time strategies;
* Template class ComObject – binds method implementation in ComRootObject to virtual table of your class by deriving from it. ComObject is the class that will be created by all factories, because without IUnknown methods your class in abstract, and cannot be created;
* Implementation class can specify various allocators to solve such problems as alignment if it contains SSE data members;
* Library also contains smart pointers to simplify the work with COM objects and work with them in RAII manner;

So, resulting module exports only single function, which is used to iterate through module map and create object instances. Module map manipulations are handled by Module class.

To centralize all object creation in the system, all creation requests are done through the Core. User module finds desired library module (by extension point or module name), receives its PluginShadow, and then sends the creation request.

Creation may be queried by specific class ID or by interface ID. In last case module will create first interface implementation class found in module map.

### 2.2.2 Using The library

In this section will be shown a full cycle of developing some library module, exporting a class, and, finally using it in other module.

Interface

Here’s some practical example of using the interface library. Below is a declaration of a simple logger interface:

#ifndef \_ILOGGER\_H\_\_

#define \_ILOGGER\_H\_\_

#include "../Core.COM/Interfaces.h"

namespace Logger

{

SCOM\_INTERFACE(ILogger, "afc7ed7f-1358-58bc-99c1-2853cb0eec62", Core::SCOM::IUnknown)

{

public:

virtual void WriteInfo(const char \*msg) = 0;

virtual void WriteWarning(const char \*msg) = 0;

virtual void WriteError(const char \*msg) = 0;

}

}

#endif

SCOM\_INTERFACE(name, guid, base) macro defines a class and binds a specified IID to it. In previous versions it required from compiler to support \_delspec(uuid( )) construction, but now in was reimplemented to use template specialization to remove this compiler-specific dependency.

Implementation

Now we will declare an implementation class of ILogger interface:

#ifndef \_CLOGGER\_H\_\_

#define \_CLOGGER\_H\_\_

#include "ILogger.h"

#include "../Core.COM/Implementations.h"

namespace Logger

{

//! Implementation of ILogger interface

/\*\* @ingroup Logger \*/

class NOVTABLE CLogger :

public Core::SCOM::ComRootObject<>,

public ILogger

{

public:

DECLARE\_IMPLEMENTATION(CLogger)

BEGIN\_INTERFACE\_MAP()

INTERFACE\_ENTRY(ILogger)

END\_INTERFACE\_MAP()

void WriteInfo(const char \*msg);

void WriteWarning(const char \*msg);

void WriteError(const char \*msg);

};

} // namespace

#endif

While writing implementation class you should always subclass Core::SCOM::ComRootObject, this class defines the implementation of IUnknown methods that will be linked to your class.

Notice the <>, it defines that we use default threading model - single-threaded. If your object will be used from multiple threads pass MultiThreadedModel as template parameter. Multi-threaded model only uses atomic operation for manipulations on reference counter and it will not protect the rest of your class.

DECLARE\_IMPLEMENTATION is used to bind the factory to your class. This one binds ComClassFactory as \_FactoryClass of your object. This factory returns new instance for every CreateInstance call. But for logging class it is better to provide one instance shared between all users, so we will change this line to:

DECLARE\_IMPLEMENTATION2(CLogger, SCOM::ComClassFactorySingleton)

This binds a specific singleton factory to Logger.

Interface map is used to form a table QueryInterface implementation, you should place there all of the interfaces you implement. In case of implementation class inheritance you can chain the QueryInterface calls by placing INTERFACE\_ENTRY\_CHAIN entry at the end of the table. Interface map also provides means to resolve ambiguous casts using INTERFACE\_ENTRY2 macro.

Exporting class from module

Now that we have an implementation class we should add it to the object map of the module. The user will create classes using their unique identifier, so we add CLSID\_CLogger GUID structure. It is often better to define all CLS\_IDs you export in separate header, so user can include it to his code without having to copy those definitions. Here how LoggerModule.cpp looks like:

#include "CLogger.h"

const Core::SCOM::GUID CLSID\_CLogger =

{ 0x862d2b3c, 0xfd06, 0x405c, { 0x8e, 0xc5, 0x75, 0x2c, 0x5a, 0x44, 0xc4, 0x7 } };

BEGINE\_MODULE\_MAP()

OBJECT\_ENTRY(CLSID\_CLogger, CLogger)

END\_MODULE\_MAP()

Creating implementation in module user

We have briefly looked on how you should implement components. Now let's take a look on them from user’s point of view.

The last step will be creation of exported object in the user’s module. To simplify DLL loading and access to the module map we will use Module class:

#include "../Core.COM/Module.h"

#include "../Logger/ILogger.h"

...

Core::SCOM::Module module("Logger.dll");

Logger::ILogger\* pLogger = 0;

module.CreateInstance(CLSID\_CLogger, Logger::ILogger, &pLogger);

if(pLogger)

{

...

pLogger->Release();

}

Note that we use module class only in unit-tests, because all classes in the engine should be instantiated through the Core.

Using objects inside your module

Sometimes you need to create the implementation objects inside the library where they were defined. There are two ways of doing it. If you need to create object, hidden behind the interface (the same case as in the user module) you should do the following:

ILogger \*pLogger;

scom\_new<CLogger>(&pLogger, UUIDOF(ILogger));

This approach creates logger instance, bypassing all overhead from using the module map. But notice, that it also bypasses object’s factory, so avoid such creation for singleton and classes with non-default factories.

The other case is when you don’t need interface, but a pointer to the implementation class, that allows calling internal methods.

CLogger \*pLogger;

scom\_new<CLogger>(&pLogger);

Interface pointer return conventions

Return conventions in COM similar to handling raw dynamically-allocated memory. Typical error in it is returning a newly-created object as return value. In this case something like this can happen:

pFooOwner->CreateFoo()->DoFoo();

In this case, Foo will be created and returned as temporary variable, then, after call to DoFoo pointer will be lost and result is a memory leak.

So here are the rules:

* Force user to accept a pointer to a variable when object is created by returning the pointer by output parameter:

Foo\* pFoo;

pFooOwner->CreateFoo(&pFoo);

pFoo->DoFoo();

pFoo->Release();

* Pass as a return value when object is cached:

pFooOwner->GetCachedFoo()->DoFoo();

These rules are also expected by logic in smart pointers discussed in next section.

### 2.2.3 Intellectual types

As you may have noticed, COM lays many responsibilities on programmers, like calling AddRef and Release at the right time and in the right place. Writing this code by yourself can be very annoying and may lead to a bunch of errors and memory leaks. To help programmer with this routine and make code more human-readable we implemented class of intellectual pointer. Let’s cover it’s usage on concrete examples.

Wrapping returned pointer

#include "../Core.COM/Intellectual.h"

...

ComPtr<ISomeObject> pSomeObj;

module.CreateInstance(CLSID\_CSomeObject, ISomeObject, pSomeObj.wrapped());

...

}

Here we creating intellectual pointer to ISomeObject interface. Note, that we give this pointer to CreateInstance with wrapped() method, this method returns the address of internal pointer inside of ComPtr class (it will work only for empty pointer, making sure you will not use it as in-out parameter). Now the pointer is safely wrapped in object, allocated on the stack, so when execution reaches "}" it will automatically call Release();.

Accessing interface methods

When you use intellectual pointer access to interface methods happens transparently, as if you was using regular pointer. But, to protect you from the errors, some trick was used that doesn’t allow calling AddRef() and Release() methods of wrapped object, so following code will not compile:

// Smart pointer initialization

...

pSomeObj->SomeMethod(); // OK

pSomeObj->Release(); // compile-time error

Wrapping an existing pointer:

If you already have a pointer that you want to wrap with ComPtr you can use Attach() method:

ISomeObject\* regularPtr;

...

ComPtr<ISomeObject> smartPtr;

smartPtr.Attach(regularPtr);

To retrieve the wrapper object use Detach() method, but note, that nor Attach() nor Detach() affect on the reference count of the object.

Type casting

Best thing about ComPtr is that it hides QueryInterface() calls:

// Initializing pointer

Core::SCOM::ComPtr<IMeshEx> pMeshEx;

module.CreateInstance(CLSID\_MeshEx, IMeshEx, &pMeshEx);

// No QI here, just upcasting

Core::SCOM::ComPtr<IMesh> pMesh(pMeshEx);

// Downcasting: calling QueryInterface() to make sure

// pMesh implements IMeshEx

pMeshEx = pMesh;

if(pMeshEx) // Checking for success

{

...

}

If the QueryInterface() fails we will have a zero pointer in pMeshEx so, using some overloaded operators, we can check if that pointer is valid. This is convenient, but watch for performance. It is better to avoid interface queries like you used to avoid dynamic\_cast in plain C++.

Avoiding old-style QueryInterface() calls with plain pointers can be also done using interface\_cast() function. Here’s the example:

ITarget\* pTarget;

interface\_cast<ITarget>(pUnk, &pTarget);

if(pTarget)

{

...

## 2.3 Reflection

Reflection is a meta-programming mechanism, sometimes also called introspection. For a good introduction to the topic see Detlef Vollman’s article on Metaclasses and Reflection[[3]](#footnote-3).

### 2.3.1 Idea and purpose of reflection

As said in the article, reflection is "looking back to oneself". That means that we keep the information about our class structure all the way to the runtime. Normally, when you compile the code in your C++ compiler, all of your class members’ names are translated to the addresses and offsets in memory. For most cases this is all what we need, but imagine a situation, when you have a script engine which should be able to execute something like this:

player = PlayerList.GetPlayerByArea(mine.Position, mine.Range);

player.Health = player.Health - 80;

Here you can see that script must gain access to such variables as CMine::Position, CMine::Range, CPlayer::Health, and CPlayerList::GetPlayerByArea() method. It looks a bit odd at the beginning, but to allow this we will implement class Class, class Method, and class Property.

Most of the managed environments like C# and Java have built-in reflection libraries, but in C++ we had to develop our own, but let's see how this looks like on Java, so we’ll know what we’re aiming for:

// Without reflection

Foo foo = new Foo();

foo.hello();

// With reflection

Class cls = Class.forName("Foo");

Object foo = cls.newInstance();

Method method = cls.getMethod("hello", null);

method.invoke(foo, null);

In the first line of reflection variant we are creating class that describes class Foo. This class called metaclass. Methods are objects too. We can get "hello" method from containing class and then invoke it with some parameters.

Assuming all written above we can define following layers:

* M0 - instance layer
* M1 - class layer
* M2 - metaclass layer
* M3 - metametaclass layer

M3 layer will not be considered further, it is used in systems that support reflection at compile time, such as UML virtual machine used in MDA.

### 2.3.2 Library Structure

Reflection library offers flexible implementation of this mechanism for C++ language.

Key features:

* Non-intrusive (reflection does not require modification of existing code, so it can be added to existing types in layer-preserving way);
* Based on C++ template meta-programming (minimal runtime overhead);
* Wide support of type conversions and parsing to ease script integration;
* Most of the data is static, minimum allocations.

Main library components are Type and ClassDescriptor hierarchies. Types are split into the following categories:

* Built-in C++ types (like int, double, etc.)
* Pointer types ({any other type}\*)
* Array types ({any other type}[{dimensions}])
* Function types (function void (\*)(int) and methods void ({class}::\*)(int) pointers)
* User-defined types (enums, classes and structures)

Plain types

Note that most of the types are not terminal (contains a reference to other types). This means that type deduction works recursively. For example such variable will have following type structure:

int\*\* array[10];

**Pointer type**

**Array type**

**dimensions = 10**

**Pointer type**

**Built-in type**

**int**

Figure 2.5 – Recursive type deduction

Usage examples:

double b = 3.14;

Type\* td = type\_of(b);

assert(tg->Tag() == RL\_T\_DOUBLE);

assert(tg->getArchTag() == RL\_ARCH\_BUILTIN);

char buf[100];

td->Name(buf, 100);

assert(strcmp(buf, "double") == 0);

assert(td->ToString(&b) == "3.14");

assert(td->TryParse(&b, "2.71"));

assert(b == 2.71);

// Foo is defined elsewhere like ‘void Foo(int a, double b)’

FunctionType\* tf = (FunctionType\*)type\_of(&Foo);

tf->Name(buf, 100);

assert(strcmp(buf, "void (int, double)") == 0);

int a = 10;

void\* args[] = { &a, &d };

tf->Invoke(args, 0);

Note that type\_of() function can return Type object of any type. Type deduction can be done with explicit type specification and deduced by passing an argument. It deduces type using templates, so type\_of() will not deduce the inherited type from base class pointer.

Each type specification supplies additional functionality. You can access dimensions of ArrayType, invoke function without knowing its true type using FunctionType, etc., just make sure to check its archtype before casting.

User-defined types

When type composition is not enough the user-defined type is used. In case of some class user may access info about class members, like fields and methods. It is also useful to get info about base classes. User-defined type serves this purpose by the means of class descriptors.

Class description can be formed explicitly or implicitly. Implicit generation hold least possible data about your type, and created only if explicit declaration couldn’t be found.

Here’s an example of explicit type description declaration:

// in header //

class CFoo : public CBase

{

public:

int a;

void DoFoo();

int getA() const { return a; }

void setA(int \_a) { a = \_a; }

};

REFLECT\_TYPE(CFoo)

// in cpp //

IMPLEMENT\_REFLECTION\_CLASS(TestClass4)

MAP\_BASE(CBase)

MAP\_FIELD("a", a)

MAP\_METHOD("DoFoo", DoFoo)

MAP\_ACCESSOR("A", getA, setA)

END\_REFLECTION()

When explicit description is declared, any type\_of() call on this type will return UserType with this description bound to it.

Continuing previous example, let’s see how to access type description:

UserType\* t = (UserType\*)type\_of<TestClass4>();

Accessor\* accA = t->FindAccessor("A");

CFoo fc;

fc.a = 1234;

int a;

accA->getValue(&fc, &a);

assert(a == fc.a);

BaseClass\* bc = t->FindBaseClass("CBase");

assert(bc->getType()->Equal(type\_of<CBase>()));

Behavioral descriptors

Besides type structure descriptors, like methods and fields, sometimes is desired to reflect the class behavior. Common examples of behavior are: possibility to create and destroy type instances through reflection, support parsing and conversion to string, custom assignment logic. For this Purposes behavioral descriptors are used. Those descriptors can be added to the type exactly the same way as structural.

Behavioral descriptors are extensively used by script engine, which will be described later, so be sure to keep reflection table up-to-date with the type definition.



Figure 2.6 – Reflection library class diagram

## 2.4 Script engine

Key library features:

* Uses developed method of Reflection-based binding generation
* Can support multiple scripting languages
* Do not require any additional binding generation tool
* Currently implemented binding generator for Python 3.1

### 2.4.1 Advantages of scripting

There a lot of advantages in using scripts for game logic:

* Engine kept as stable game-independent system
* Scripts prevent game logic leaks to lower system levels
* High-level script languages are easy to learn and use
* Scripts can be modified without need to recompile the program

### 2.4.2 Script integration

The hardest part of script integration is binding it to C++ code of the engine. This means that all functions that should be accessible to the script should be explicitly exported to it. This is a huge task and a lot of tools are created to solve it. Some of them use additional C++ compilation to create proxy-objects from C++ code. Some use C++ templates to create bindings at compile-time. The best method should fit following criteria:

* Shouldn’t complicate the build procedure
* Shouldn’t require modification of existing code
* Minimize the amount of “glue-code”

The solution was to use existing reflection mechanism to generate script binding at the run-time. Bindings can be generated by iterating through all registered types and processing of type structure. This way all information that is handled by reflection (data members, accessors and methods) can be easily exported to scripts.

### 2.4.3 Scripting language

Different scripting languages provide different functionality, so choosing one among them is a hard and responsible task. The situation is even more complex if some developer team prefer or more familiar with one specific language.

Because Sekai uses reflection to create script bindings, it is not limited to some specific script language like other engines. Multiple scrip system implementations can be provided. The script engines only differ by binding procedure and script management, so it is not costly to have multiple scripting systems (this is impossible for other engines that explicitly create script-specific bindings).

Currently script binding is implemented for Python 3.1 language and includes full support of reflected constructs (methods, fields, accessors, parsing etc.). Binding is bidirectional, this means that you can either access and subclass C++ types from Python, or manipulate Python types from C++ code. Further in this chapter will be covered library implementation and common usage approaches.

### 2.4.4 Python binding implementation

Python/C API provides means for extending Python with user-defined types by adding extension modules. This mechanism is primarily oriented on Python extension using small dynamically loaded libraries that contain the new types and their behavior logic. But in the case of game engine script interpreter should be extended and embedded simultaneously. This means that extension is not composed in a stand-alone library but instead is attached dynamically to the embedded interpreter. Using the reflection, extension not only attached, but also created at the runtime.

Exporting types to script

Consider the following class with corresponding reflection table:

class Player : public PlayerBase

{

public:

const VML::Vector3& getPosition() const { return mPosition; }

void setPosition(const VML::Vector3& pos) { mPosition = pos; }

private:

VML::Vector3 mPosition;

};

IMPLEMENT\_REFLECTION\_CLASS(Player)

MAP\_BASE(PlayerBase)

MAP\_ACCESSOR("Position", getPosition, setPosition) MAP\_CREATABLE2(new\_player, del\_player)

END\_REFLECTION()

This is a pretty simple class, inherited from PlayerBase. Reflection table is simple too. The only uncommon thing here is MAP\_CREATABLE2 macro. This macro is used to override default construction logic of reflection library. We have to do this because Player have a member of type VML::Vector3 that requires a specific memory alignment, so it can not be allocated using simple new. Now we have a test class, next step is to make it accessible to the script. This is only a matter of one line of code:

pScriptManager->ExportType(“sekai\_test”, type\_of<Player>());

Here’s the steps that covered behind this call:

1. type\_of returns the Type object that describes the Player class;
2. Script manager checks the module registry searching for “sekai\_test” module;
3. If such module does not exist it creates an empty module with this name and registers it in Python’s import table, so it can be accessible to the scripts;
4. Script manager recursively creates bindings for the base classes of our type;
5. The new type description object is created, describing the general type behavior, like allocation/destruction logic, property getter and setter callbacks, str() operation, assignment and comparing operations. The set of operations depends on behavioral descriptors of type defined in reflection table;
6. When type is created, script manager iterates through the type’s methods creating the method proxies in Python. Method proxy contains a C++ function callback and a method ID in the reflection table, so it can be found fast in the invokation time;
7. New type is inserted to module table and now accessible to the scripts.



2.7 – Script system overview

Example of the script:

>>> import \* from sekai\_test

>>>

>>> p = Player()

>>> assert(issubclass(p.\_\_class\_\_, PlayerBase))

>>>

>>> pos = Vector3()

>>> pos.x, pos.y, pos.z = 1, 2, 3.5

>>> print (pos)

1.0, 2.0, 3.5

>>> p.Position = pos

>>>

>>> print (p.Position)

1.0, 2.0, 3.5

Note: the base classes are exported automatically, but argument and return types (like Vector3 in this case) should be exported explicitly.

As you can see, the exported object is a fully-valid Python type, so nothing stops you from subclassing or extending it.

To implementd argument and return value passing between two languages a special marshalling mechanism. It uses the type information of the reflected methods to corretly convert the types before invoking them. Because script may invoke the functions that return values by pointer the exported type instances are separated in two groups:

* Owned – instances created inside the script and with lifetime controlled by it;
* Shared – instances that were passed to the script from outside

Interacting with script from C++

The scripting system will not be complete without the means to access the scripted types and variables from the C++ code. This mechanism is also powered by reflection and uses the same marshalling as the rest of the library. Here is the usage example:

# Script

>>> class TestClass(object):

>>> def \_\_init\_\_(self):

>>> self.mData = 0

>>>

>>> def setData(self, d):

>>> self.mData = d

>>>

>>> def getData(self):

>>> return self.mData

// C++

IScriptObjectInstance\* tc1;

mgr->CreateObjectInstance("py\_classaccess.TestClass", &tc1);

int d = 10;

ScriptInvArg args[] = { &d, type\_of<int>() };

int r = -1;

ScriptInvArg ret = { &r, type\_of<int>() };

tc1->InvokeMethod("getData", 0, 0, &ret);

assert(r == 0);

tc1->InvokeMethod("setData", args, 1, 0);

tc1->InvokeMethod("getData", 0, 0, &ret);

assert (r == 10);

tc1.Release();

This mechanism can be used to create Python-C++ proxy classes that will dispatch calls to script, allowing creating and using classes whose implementation is defined in script files.

# 3 Service libraries overview

Core layer also contains several service libraries, in this section they will be covered briefly.

## 3.1 Logging library

Logging library provides convenient output of debug information and useful debug functionality.

Key features:

* Thread-safe logging
* Compile-time and runtime logging policies
* Exception stack trace generation (Windows-only)
* Mini-dump generation on fatal errors (Windows-only)
* Accessible from all modules through global environment
* Listener facility

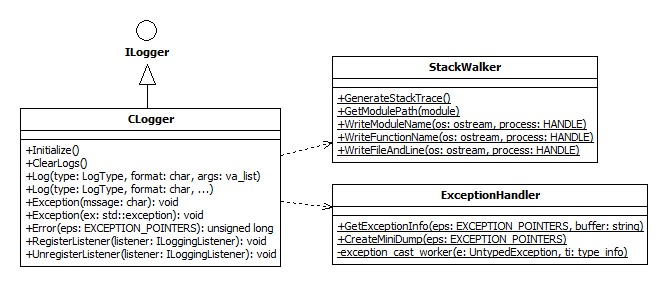


Figure 3.1 – Main logging library components

## 3.2 File system library

File system library provides cross-platform way to access file system resources.

Key features:

* Object model of file system
* Resource tree is built incrementally as user traverses it
* Support of asynchronous operations and completion callbacks
* Transparent manipulation with archives
* File adapter facility provides flexible way for all modules to register their parsing classes that provide access to object model of various types of files. Built-in adapters are XML and INI-file adapter.



Figure 3.2 – File system library class diagram

## 3.3 Math library

Math library is a greatly extended version of Sony Vector Math Library. It provides optimized math routines for SSE and SPU instruction sets. Only AoS (Array of Structures) data format is used.

Library includes following types:

* Point (2 and 3-dimensional)
* Vector (2, 3 and 4-dimensional)
* Quaternion
* Matrix (3x3, 3x4 and 4x4)

Because SSE and SPU types have some strict alignment requirements, library also provides specific allocators that compatible with STL containers and SCOM library.

## 3.4 Time library

Time library consist of two main components: timers and interpolation framework.

### 3.4.1 Timers

This library provides means of time source abstraction and clock manipulation.

Time source provide methods to get current number of ticks, and resolution – number of ticks per second. Time source implementation can vary. It can be bound to system clock or even be driven manually.

Clock objects used to measure time between some intervals, frames for instance. Clock can run in two different modes:

* It can be bound directly to time source. Then, on every call to Update(), it will read the current tick number and translate it to number of seconds since last call to update;
* It can be bound as a child of other timer. In this mode clock can’t be directly updated, it only receives the interval from parent timer, applies scale to it, and passes new value down the clock tree. This mode allows inherent control of the time scale.



Figure 3.3 – Clock and TimeSource interfaces

### 3.4.2 Interpolation framework

Interpolators provide convenient way to create and store objects’ animation. Interpolators are commonly used to animate the objects and camera in the cut-scenes using a set of predefined key frames.

Interpolation theory

Simplest type of interpolation is linear interpolation. Linear interpolation uses line equation to calculate the data between the key frames. This approach is fast, but hardly suitable in most cases, because produces a sudden noticeable changes in motion direction. Below is the illustration of how camera movement will look like using the linear interpolation method.

Figure 3.4 – Linear interpolation of position

To achieve smoother animation without the increasing the number of key frames high-order interpolation techniques is used. These methods achieve C1 continuity using parametric curve equations. Commonly used are Hermite curves and Cubic Splines. Along with position data Hermite curves use a predefined tangent vectors. These types of curves are mostly used in the graphic packages where best control over the curve is desired. Cubic splines also use a set of tangent vectors, but these vectors are computed automatically using only position data.



Figure 3.5 – Cubic Spline interpolation of position

Cubic splines are accurate but require more memory for computed tangents. When the quality is not very important a more robust Catmull-Rom splines can be used. These splines compute tangents at interpolation time, using only positions of two neighboring points.

For full motion animation interpolation of position is not enough. Key frames can also be used to interpolate the orientation. For fast interpolation Sekai engine stores the orientation in Quaternions.

Interpolation library

Currently interpolation library implements following interpolators:

* Vector interpolation:
  + Linear
  + Catmull-Rom spline
  + Cubic spline
* Orientation interpolation:
  + Linear
  + Spherical
  + Adaptive (linear for small angles, spherical for big)

Interpolators are implemented as independent library, based only on Math module in sake of reusability. As interpolation criteria interpolator receives floating-point value typically in range [0; 1]. For values beyond this range different time modes can be used:

* Clamp (one animation cycle)
* Wrap (cycled animation)
* Mirror (cycled animation repeated in forward and then reverse order)

Engine integration

Framework provides means to bind different types of interpolators to the values inside of some object. Was decided to use reflection mechanism to access the data to allow binding to raw values as well as to values hidden behind get/set accessors, and make library non-intrusive, not requiring from objects to implement some specific interface.

To attach an interpolator to desired object (typically entity component) an InterpolatorBinding is used. This object contains an interpolator and reflection data accessor. When binding is registered in AnimationController it will automatically update data every frame until detach criteria will be met.

# 4 Engine layer

When Core layer contain the service libraries and general functionality, Engine layer is more game-oriented. Now it contains:

* Common graphics engine interfaces and objects
* Graphics engine implementation for DirectX 9
* Input library
* Main engine module

## 4.1 Graphics engine

In this section will be covered all design decisions similar for all graphics engine (GE) implementation. Engine.Graphics library provides common interfaces that should be implemented by all engines.

Key features of subsystem:

* Shader-oriented
* Support of shared resources
* Support of resource archives
* Asynchronous resource loading
* Procedural mesh construction support
* Dynamic shader binding through description files
* Flexible system of material callbacks
* Debug information rendering support (lines and bounding volumes)
* Multiple render target rendering

### 4.1.1 Design approach

There are two general approaches to graphics engine design:

* High-level interface
* Low-level API wrapper

Graphics engine as high-level interface

First approach means implementing in GE interface all of the high-level rendering functions, for example drawing of sky box, cloud rendering, etc. Implementation of these functions is done directly in rendering API terms.

Advantages of this approach are:

* Easy to implement
* Easy to use

Drawbacks are:

* GE becomes application-specific, so almost no code reuse possible between projects
* Hard to maintain multiple implementations for different APIs

Graphics engine as low-level API wrapper

Second approach assumes creation of low-level API wrappers, what means that all DirectX and OpenGL functions are generalized, and a single superset of functionality is created.

Advantages of this approach are:

* Full graphics engine reuse between projects
* Implementation defined in one place and works for all supported APIs
* Scalability

Drawbacks are:

* Hard to design and implement

Because current engine is designed as middleware for various games it can’t contain any application-specific functions, so the second approach was chosen.

Also was decided to keep rendering interface as simple as possible, that’s why it acts much like a GDI. Engine don’t know about objects’ positions (just transformations), do not do any optimizations (like depth, geometry, and material sorting). Those functions are became responsibility of higher layers.

This approach helps to keep minimum amount of API-dependent code, without any loss in flexibility.

### 4.1.2 Renderer interface

For maximum scalability with minimal set of wrapped functionality was decided to make graphics engine shader-oriented. This means that no fixed pipeline functions are supported. Low-level state manipulations are preferred to be inside of the effect files. The high-level state objects will be added to support state managing in scene graph and allow rendering optimization by state sorting. Below is the representation of renderer interface.



Figure 4.1 – Renderer interface

Most of functions are self-explanatory, details about texture sets and material callbacks will be covered in later chapters.

### 4.1.2 Resource managers

All graphics resource creation is done through resource managers. All resources are split into three categories: geometry, surfaces, and materials – there’s one specific manager for each. Now each resource created as shared. This means, if in one level there are several objects with the same model, all those objects will share the geometry, texture, and material info. Later sharing policies will be added to allow dedicated ownership over the resource. This is useful for software morphing animation, when all animated objects should have own geometry.



Figure 4.2 – Resource managers

Resources are looked-up by special URLs (Unique Resource Locators). File access is done using FileSystem library, so it is possible to gather resources in archives.

For details on resource-specific creation logic see resource topics.

### 4.1.3 Resources

In this topic all resource types will be covered in more detail. One common thing for all resources is an additional reference counting (beside reference counting of COM object). This reference counter determines resource usage. That means resources act like proxies to themselves, when no one uses them they unload all memory-consuming data, leaving minimum of info to make a load on further demands. This helps in creating big worlds, when many objects will not be seen at once and their resources can be unloaded.

Meshes

Mesh is a compound object that contains multiple geometry objects, called subsets. All data about vertices and indices of all subsets are stored in two corresponding buffers. Data inside those buffers don’t overlap between subsets. It is sorted and decomposed to improve rendering speed.

Geometry objects are graphic primitives – smallest thing that can be drawn. Geometry contains reference to parent Mesh and its bounds inside shared buffer. All geometry objects can have an attached material object and texture set. Note, that neither Mesh nor Geometry provides high-level rendering methods. The responsibilities for material, texture, geometry managing and directing of rendering process are laid on user. With this low-level access user can implement various techniques to improve rendering speed, like geometry and material sorting.

Meshes use flexible vertex format, when vertex data is defined in special description table with usage, data type and offsets.

Mesh is loaded through special mesh description files, which specify geometry subsets, their textures and materials. In future it is planned to add LOD (Level of Detail) support.



Figure 4.3 – Graphics engine resources

Textures

Textures are two dimensional surfaces with encoded color values. Format of the texture data is defined by EBufferFormat enum. There are two special texture types: dynamic textures and cube maps. Cube map is a texture composed from six rectangular surfaces, and commonly used for skyboxes. Dynamic textures are those which allocated in memory, not loaded from files, and commonly used as additional render targets. Only dynamic textures can be queried for RenderTarget object.

For dynamic shader binding textures can be organized in texture sets. Texture set assigns to each texture a specific ETextureType. This allows linking some texture to shader, when it queries, for example, normal map texture.

Materials

Material is a wrapper on effect file which additionally provides dynamic parameter binding. Parameters are associated using the material description files, which define a specific EMaterialParam for each external shader variable.

Now only single technique of effect file is supported, but later techniques will be used to implement fallback mechanism for different graphics quality settings.

Material callbacks

When geometry with some material is ready to be drawn, material is responsible for refreshing the parameters of the shader. This is done by using the material callbacks. Material callbacks provide methods for querying the parameters by their EMaterialParam.

Default callback of the engine uses current data from transformation stacks and texture sets to fulfill texture and transformation request. All additional data, light position for example, can’t be gathered on graphics engine’s level. That’s why support of callback chaining is implemented. Chaining allows creating of “Chain of Responsibility” for material parameters, when low-level queries can be handled by graphics engine, and all not satisfied queries will be transported further through chain to higher level.

### 4.1.4 Rendering process

In this section rendering process will be described step-by-step, to clarify the dependencies between components.

To render some geometry to the screen user must:

1. Signal frame rendering start by calling IRenderer::BeginFrame()
2. Clear surfaces (depth/z-buffer/stencil)

Note: because stencil and z-buffer is basically one buffer, clearing both of them may increase performance by eliminating partial data access.

1. Set the world, view, and projection transformation matrices in corresponding matrix stacks
2. Set the current texture set (needed for texture lookup by default material callback)
3. Set the material
4. Set the geometry buffer
5. Update material bindings
6. Begin material sequence
7. Iterate through material passes
8. Draw geometry between materials BeginPass() and EndPass()
9. End material sequence
10. Remove transformation matrices from the stacks
11. Call IRenderer::EndFrame()

Changing render targets and MRT rendering

In the rendering process described above, renderer uses the default render target (RT) – back buffer. Such complex effects like deferred shading may require more than one render target to be set in a single rendering pass. Similarly to simple changing of render target, this can be achieved by using IRenderer::setRenderTarget() method which can set render target with specific index.

For example, if in your pixel shader you have an output type defined as:

struct PS\_Out

{

float4 Diffuse : COLOR0;

float4 Depth : COLOR1;

};

Each field of this structure outputs to different render target, Diffuse – to RT with index 0, Depth – to RT with index 1.



Figure 4.4 – Color-based edge detection as post-processing effect

When implementing post-processing effects it is common to use a buffer “ping-ponging” technique. In this technique commonly only two off-screen textures are used. At each moment one of the texture are the source, and second is target.

For example, consider Gaussian blur effect. It can be efficiently implemented in two passes – vertical and horizontal blur. First we render initial scene using first texture (T1) as target. Then we apply horizontal blur while rendering full-screen quad to second texture (T2). On the third step T2 becomes a source, and back buffer a target, while rendering image with vertical blur.

Effect 1

Effect 2

Scene

…

**T1**

**T2**

**T1**

**BB**

Figure 4.5 – Buffer “ping-ponging”

## 4.2 Input library

Now input library is based on DirectInput and supports mouse and keyboard polling.

[Library will be extended for various controller support, documentation pending]

## 4.3 Main engine module

Key features:

* Clean interfaces
* Strict separation of concerns between subsystems
* Dynamic system composition
* Component-based entity structure
* Flexible game logic system with update scheduling
* Hierarchical transformations
* Integration with NVIDIA PhysX

### 4.3.1 Module goals

Main goals of Engine module are:

* Provide means of application-independent managing of data
* Combine all subsystems and drive their interactions
* Supply “building blocks” for game entities

### 4.3.2 Engine object

IEngine is a facade interface to all subsystems. It is created by Core on system startup because it’s subscribed to the core::startlisteners extension point. Then engine uses its own extension points to create suitable implementations of main subsystems like renderer, input library etc.

Engine’s global environment

Engine uses Core’s global environment extension mechanism to simplify the access to different subsystems. Now it provides access to following objects:

Table 4.1 – Main objects of Engine’s global environment

|  |  |  |
| --- | --- | --- |
| Name | Interface | Description |
| Engine instance | IEngine | Engine object instance |
| Main loop | IMainLoop | Used to notify frame listeners |
| Input device controller | IInputController | Polls the input devices |
| Component registry | IComponentRegistry | Used to create entity components exposed by different modules |
| Scene manager | ISceneManager | Used to create new entities |
| Physics manager | IPhysicsManager | Used to execute binding between engine’s and simulation objects |
| Transform graph | ITransformGraph | Used to create transformation hierarchies |
| Logic controller | ILogicController | Drives the process of “Think” notifications of logical components |
| Renderer | IRenderer | Facade interface to rendering API abstraction |
| Material manager | IMaterialManager | Manages the material objects |
| Surface manager | ISurfaceManager | Manages the textures and render targets |
| Geometry manager | IGeometryManager | Manages the geometry |
| PhysX SDK object | NxPhysicsSDK | PhysX SDK object |
| PhysX scene object | NxScene | PhysX scene object |

Because extension system prohibits the environment substitution, only extending, all previous services accessible too.

### 4.3.3 Entity structure

Heart of any engine is its approach to game object structure. In Sekai game objects called Entities, and component-based approach is used.

Traditionally, game entities were represented as a deep class hierarchy. In frequently modified game engine this approach usually led to huge class hierarchy with blob interfaces and repeated functionality. This approach is also more application dependent, because it is hard to develop reusable engine components based on inheritance.

In Sekai engine aggregation is used instead of inheritance. Entities are described not in IS-A, but in HAS-A (has some functionality or data). This means that entity is fairly simple object that has a name and a set of some component. The components are those who define an entity behavior.



Figure 4.6 – Example of entity composition

From fig. 4.4 you can see that component-based system prevents duplication of functionality. If the object not supposed to have some abilities corresponding component just will not be added.

Aggregation relationships are more flexible than inheritance and can even be changed dynamically.

### 4.3.4 Built-in components

Engine provides four major built-in components: Appearance, SpatialNode, PhysicalBody, and Logic. Those components implement most of the application-independent functionality. They are integrated into the engine and work by the same rules as user-defined components will.



Figure 4.7 – Class diagram of built-in components

SpatialNode

This component is used to identify the position of the object in the world. The position is not given in the world coordinates, transformation hierarchy used instead. That means that all objects form a transformation tree where parent of some branch influences on the position of its children. This allows creating complex objects without controlling position manually.

Transformations are handled by TransformGraph object, which updates global positions every frame, when all changes were made. After this update the final transformation matrix can be used to draw objects.

Appearance

This component is responsible for how entity will look like when it is drawn on screen. Now it is only defines mesh to be drawn, but further the support of state objects will be added.

PhysicalBody

This component binds entity with its representation in the simulation engine. Binding can be one of two types: driven by engine (kinematic body), or driven by simulation (rigid body). Binding type determines the direction of position updates between two systems.

When entity is controlled by user (directly and/or by animation) it should have a kinematic binding type, so that engine will update its position in physics simulation.

When entity should respond to collisions according to its physical properties rigid binding type is used. So, each frame, the engine will fetch the new position from simulation system.

Binding are controlled by PhysicsManager object.

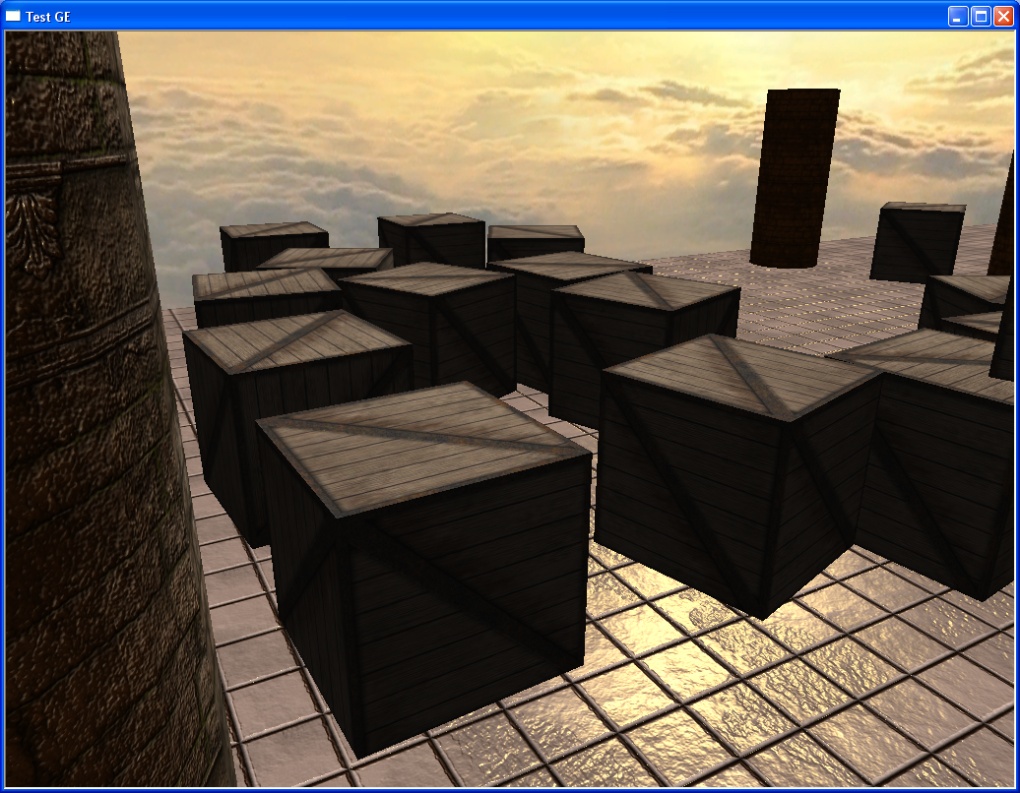


Figure 4.8 – Binding example: crates are rigid bodies, player is kinematic

Logic

Logic components allow entities to execute some game-specific actions. Those actions should be performed in response to notifications, which come to the component by calling Think() function. Although Logic is stated as built-in component, it is not fully-implemented. Only base objects are provided to simplify the definition of own components.

CCmpLogicBase controls component registration in LogicController and manages the ThinkContext.

Using the ThinkContext user can vary the time and frequency of notifications from controller (now implementer: THINK\_EACH\_FRAME, THINK\_SCHEDULED, and THINK\_NEVER). Scheduled updates allow setting the time of the next notification explicitly.

### 4.3.5 Creating own components

In order to add some component type user should implement IComponent interface and expose this implementation from he’s module. At the startup he should register the component using ComponentRegistry object using component class ID and he’s plug-in shadow. When this is done, ComponentRegistry can be used to create instances of this component and bind them to entities.

Component creation is driven by component description objects, which define component ID to be created, and a specific set of parameters. Description objects are passer to IComponent::Bind() function, along with the target entity, to fully initialize the component.

Type casting

When it is necessary to check that component implements some specific interface the built-in SCOM functions are used (QueryInterface or smart pointer casting). Type casting can be useful when some component depends on other component of the same entity, and its existence should be checked at the binding step.

## 4.4 Logic Graph

System is in development

All logical components can be represented as objects that have a specific set of parameters, react on some notifications, and produce own set of events in response. Game logic can be described by a set of logic graphs, which define interactions between entities. Logic graph is an instance-level structure.

**BossEntity**

Died

ID

**MoveToPosition**

Finished

Entity

…

**MissionObjective**

Completed

Failed

Figure 4.9 – Logic graph example

The structure of graph node can be held in special entity component. For complex entities, which have more than one component, node structure can be generated at component link time.

Because nodes in the graph are game-specific, only basic node types will be implemented in the engine (like area and proximity triggers etc.), other node types will be implemented by user and loaded as plug-ins.

Event-based nature of the system allows implementing all node logic in script files.

## 4.5 Shading Graph

System is in development

The shading techniques become more and more complex over time. The introduction of multistage rendering is made unification of rendering process a hard task. The idea behind the shading graph is to provide an object-oriented way to describe rendering process of the scene once and use it throughout all runtime. Key components of the shading graph are render stage and rendering step.

Render step is a smallest element of rendering pipeline, a single draw call. Rendering step requires geometry to render, effect to use, a pass to execute, an input texture set, and an output buffers (render targets).

Render stage consist of multiple render steps and used for high-level rendering control. Stages are executed sequentially. Render stage has a geometry source, which supplies a geometry that will be rendered. Typically it is a culling mechanism that determines a set of objects visible from current point of view. Render stage perform a sorting of primitives to speed-up the rendering.

**Effect:Paass0**

**Textures**

**G-Buffer**

**Geometry**

**Step 0**

**GEOM**

**SRC**

**Stage 0**

**Effect:Paass0**

**Back Buffer**

**G-Buffer**

**Geometry**

**Step 0**

**Stage 1**

**QUAD**

Figure 4.10 – Deferred shading example

Example of the frame rendering using deferred shading algorithm described by a shading graph:

1. Executing Stage 0:
   1. For each geometry from the geometry source
   2. Render a geometry to the G-Buffer (a set of render targets);
2. Execute Stage 1:
   1. Render a full-screen quadrilateral using G-Buffer as source texture and Back Buffer as render target.

If the user’s hardware doesn’t support a MRT rendering than in the first stage we will be using multiple steps, each of which will output to a different render texture.

# 5 User Interface Layer

Although functionality in Engine layer is more than enough to implement any 2D and 3D graphics, it’s convenient for user to have some primitives to start designing he’s own user interface.

UI layer now in early stage of development so now it only contains following components:

* Library for rendering of vector graphics
* SVG-parser and converter
* Font engine

## 5.1 Vector Graphics Library

Top goal in engine design always was to achieve maximum agility and scalability of the architecture. From the very beginning was decided to write as extensible code as possible, with orientation on further growth. So, when the time came to the GUI system design, raster graphics was dismissed almost immediately.

GUI is no more just an overlay above 3D game viewport. It should be integrated with the game process. That means that GUI controls can be placed as in viewport space as in game world. Computer displays and holographic screen should be able to use the same UI system as main game interface.

Now there is no non-commercial library suitable for real-time VG rendering. Closest match are OpenVG and related projects, but none of those is suitable for games, because they either used software renderer, or OpenGL by-pixel rendering. Commercial libraries are inaccessibly expensive.

In Sekai engine implemented our own vector graphics rendering library.

Advantages and drawbacks of VG

Both are pretty obvious, main advantages are infinite scaling, without loss of details. Also, because VG is described with geometrical primitives, it consumes significantly less memory than raster one. But drawbacks are serious too. For now there is no widely-used hardware support for VG, so raster graphics will surely be faster than VG. But this is not for long. VG is gaining strength, example are WPF and upcoming Direct2D.

Requirements

Formalized requirements for the library are:

* Should fit the game’s FPS limit
* Should load the GPU as much as possible
* Customizable shader compatible
* Low level rendering should be isolated for later implementation using Direct2D or something else.

Rendering pipeline

1. Initial data is buffer of points and associated commands. Commands describe what primitive will be drawn (line / Bezier curve etc.) and which mode of drawing is active (line joint and ending styles etc.). Coloring modes are also supported (with alpha blending).
2. This data is wrapped by Path class. Path always holds the initial data, so that resulting image can be reconstructed with new quality at any time.
3. Before first frame, and later on demand, VG data is transformed by ApproximationUnit that approximates curves with lines with given tolerance.
4. Line buffer then passed to fill and stroke tessellators. They convert lines to set of triangles.
5. After those steps we already have all data for rendering, which is controlled by Layer class.
6. VG library uses special technique for rendering of polygons. Because polys, defined by user in graphics editor can be concave, and task of tessellation of concave polygon is computation intense, library uses stencil buffer rendering trick. Also, as in other VG libraries, fill color of multiple intersecting polys in single path will be XOR’ed (filling of one area for two times will produce empty region).
7. Paths within layer are rendered front-to-back to reduce overdraw, and with no depth checking.

The VG library uses the renderer interface and special types of materials for this complex rendering technique. This is another proof of the material system versatility.

Supported primitives

Main idea is that each primitive requires defined number of points. First point and first command in the Path is always PC\_MOVE, which sets the initial point of the path. Next let’s say we want to draw a quadratic Bezier curve, it requires two anchor points and one reference point, three in total. But remember that first point is already set, so only two points should be supplied. Now you should have the idea how points and command are matched.

Primitives:

* Line
* Quadratic Bezier
* Cubic Bezier
* Quadratic and cubic smooth Bezier
* Arc

Performance

Even first version of the library showed suitable FPS in fairly complex image (tested on SVG image transformed to library format). FPS statistic below is taken when image was recomputed each frame. In normal situation recomputation will be required only when you scale the image, so that curves can be approximated with different tolerance. In other time, rendering will take just about the same time as drawing of regular mesh (but with small overhead for stencil).

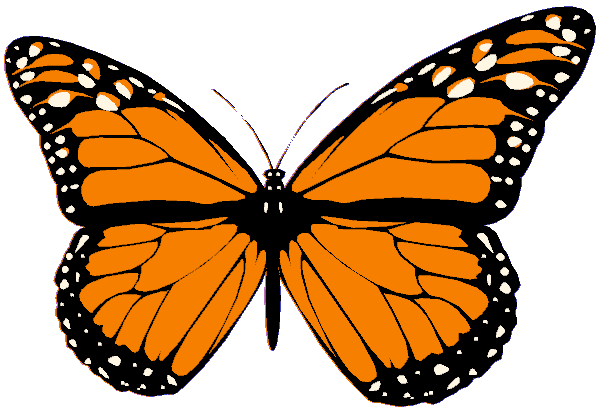


Figure 5.1 – Example of SVG-image rendered with VG library (2,000 triangles)

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Tolerance | Triangles | Total frame time | Approximation time | Rendering time |
| Minimal | 7,700 | 14,1 ms | 3,5 ms | 10,6 ms |
| Average | 2,000 | 4,3 ms | ~ 1 ms | ~3 ms |
| Big | 1,300 | 2,7 ms | - | - |

Sample image consist of 1,200 Bezier curves, so, near maximum tolerance, most of Bezier curves were approximated with single line.

VG library approximation and tessellation algorithms are very suitable for parallel computation, so performance can be easily increased even more.

## 5.2 Font engine

Current implementation of font engine is fully based on VG library. Using a FreeType library it loads a font files an extracts path data in form of vector graphics primitives (quadratic and cubic Bezier curves). Then this data is transformed to a VG library format and converted to the triangle mesh.

Because this conversion is computationally-intensive, and the number of triangles needed to approximate a text is extremely high, this system can be used only for rendering a small amount of high-quality text and in cases when some 3D algorithm should be applied to text mesh.

Further work will be concentrated on creation of abstract font interface. This interface will allow user to choose to use a high-quality text approximated with a triangle mesh, or a simple image-based text when high performance is needed.

# Appendix A: Setting up the environment

This chapter discusses installation of all prerequisites needed to build the engine. Notice, that following requirements are those needed to build the engine’s main modules, each plug-in may have its own additional library dependencies.

## External dependencies

|  |  |  |
| --- | --- | --- |
| Layer | Dependency | Description |
| Core | [STLPort](http://www.stlport.org/) | Most successful implementation of C++ standard library |
| [Boost C++](http://www.boost.org/) | Collection of various C++ utility libraries.  Usage:   * Unit-testing framework * File system path abstraction * Date-time library * Regular expressions |
| [TinyXML](http://www.grinninglizard.com/tinyxml/) | Used for XML parsing (sources included with engine) |
| [ZLib](http://www.zlib.net/) | MiniZip extension is used for zip-file reading (binaries included with engine) |
| [Python](http://www.python.org) | Script binding system is compatible with Python 3.1 |
| Engine | [DirectX SDK](http://msdn.microsoft.com/en-us/directx/aa937788.aspx) | Used in stock graphics engine implementation |
| [Sony Vector Math Library](http://sourceforge.net/project/showf...roup_id=147573) | Provides portable optimized math functions for SSE and SPU instructions (greatly extended, included in source distribution) |
|  | [NVIDIA PhysX](http://developer.nvidia.com/object/physx_downloads.html) | Powerful physics simulation library |

## Building STLPort and Boost libraries

In our engine we use STLPort and Boost C++ library built on top of it. Here’s the build instruction for those libraries using MSVC9 compiler:

Building STLPort

* Download latest version of STLPort (tested with 5.2.1)
* Unpack the archive
* Note: STLPort uses some tricky allocators that cause Visual Studio to detect a lot of false memory leaks in it. Those leaks are not a problem by themselves, but they can obscure the real ones, so while debugging you better use STLPort configured for simple allocators. To do this go to stlport/stl/config/hosts.h and uncomment \_STLP\_LEAKS\_PEDANTIC and \_STLP\_USE\_NEWALLOC. Current STLPort version fails on x64 build with this setting.
* Run configure.bat msvc9 --with-dynamic-rtl --extra-cxxflag /Zc:wchar\_t
* Open VS console and move to %STLPort\_Folder%/build/lib and run nmake clean install (to build for x64 platform run x64 compiler console)
* Add lib and STLPort directories to VS
* When linking dynamically make sure dlls in bin directory are visible to linker (use environment variables or just copy them to system directories)
* Now you can delete all object files in build/lib to save some space
* For convenient use download [visualizers](http://stlport.svn.sourceforge.net/viewvc/stlport/trunk/STLport/etc/autoexp.dat?revision=HEAD) and add them to autoexp.dat file of VS

Building Boost C++

Boost library has to be built on top of STLPort, so this will require some more steps than normal build, STLPort-specific steps will be marked with (\*) so you could use this manual to build Boost even without it.

* Download latest version of Boost and BJam
* Unpack boost to C:\Program Files\Boost or elsewhere
* Compile BJam using .bat file and place the .exe file to C:\Program Files\Boost\boost\_1\_xx\_x\
* (\*) Find in boost package file called user-config.jam and copy it to root boost folder near bjam.exe
* (\*) Edit this file so that STLPort option was uncommented and specify include and library folders, it will look like this:

# Configure specifying location of both headers and libraries explicitly

using stlport : : D:/Programming/Soft/STLport-5.2.1/stlport D:/Programming/Soft/STLport-5.2.1/lib/x86 ;

* Open console in this folder and execute one of following (to build boost for x64 platform run bjam with VS x64 compiler command prompt):

#x86 with STLPort

bjam --toolset=msvc-9.0 stdlib=stlport --without-math --without-wave --without-python --build-type=complete --user-config=user-config.jam

#x64 with STLPort

bjam --toolset=msvc-9.0 stdlib=stlport --without-math --without-wave --without-python --build-type=complete address-model=64 --user-config=user-config.jam

#x86 standard

bjam --toolset=msvc-9.0 --without-math --without-wave --without-python --build-type=complete

#x64 standard

bjam --toolset=msvc-9.0 --without-math --without-wave --without-python --build-type=complete address-model=64

* Wait for an hour until build is finished
* Now you should have bin.v2 folder in your boost directory
* Create lib(\x86,\x64) folder and copy all the .lib and .dll files from bin.v2 from it
* Unlike STLPort, we use only static linking to boost, so all dynamic builds are useless, I will modify the build string soon so that it will not produce dynamic libraries and builds with static CRT linking.
* Go to Visual studio and add directories for:
  + Headers: C:\Program Files\Boost\boost\_1\_35\_0\
  + Libs: C:\Program Files\Boost\boost\_1\_35\_0\lib\

As soon as all libraries installed you can compile the engine and proceed to documentation.

# Appendix B: Thoughts for the future

## Greedy Material System

What is material really? In the books it is always described as properties of a surface related to how it reflects / refracts / absorbs / emits / etc. light. Generally saying it is how the surface looks like. Textures, normal maps, specular power, all this properties used to render a geometry as realistic as possible. So the common practice is to assign a material to geometry, to specify how it should be rendered. Material always includes the shader, which is capable to combine input data and produce the resulting image in render target.

Real life is different. There are no separate shaders used to “render” different objects. All objects comply with the same rules and “rendered” similarly. A God-shader maybe :)

This means that the looks of the object should be determined only by its properties, not by the shader it uses. Ideally, system can have a single shader that knows how to draw any object in the scene (normal-mapped, with parallax, etc.). This is possible, but for sure not efficient. So in real system shaders can be considered only as means to simplify rendering process when object provides simple properties.

This concept means that shaders can be completely separated from the objects, they can be thought of as “rules of rendering the whole world” kept separately in the graphics engine. The system called greedy, because it gathers as much as possible info about the objects, before choosing the appropriate shader.

For example, those objects, which only provide color and specular power, are rendered with a “solid color” shader. Those which provide a diffuse texture will use “textured” shader. Same thing for normal-mapped, parallax etc.

Design features:

* Separation of shaders from geometry;
* Design consistency;
* Ability to improve the whole game by switching to another shader library (even after release);
* Separates shader description from actual parameters;
* Easy to implement material level of detail (material fallback with increase of distance).

Design flaws:

* Material sorting? Solution: determine the final shader first, than sort by it;
* How to chose shading type (flat, Gouraud or Phong)? Solution: another set of parameters – render states;
* What about special objects like GUI? Solution: give ability to override the shader picking system and specify shader explicitly.

1. Erich Gamma “Contributing to eclipse - principles patterns and plug-ins” [↑](#footnote-ref-1)
2. Don Box “Essential COM” [↑](#footnote-ref-2)
3. Metaclasses and Reflection in C++ <http://www.vollmann.com/pubs/meta/meta/meta.html> [↑](#footnote-ref-3)