程序编码规范

# 文档说明

本文档包含本软件制作过程中，程序编码方便的准则，规范。请编码人员认真阅读。

# 命名规范

## 文件名命名

文件名应全部采用英文小写命名，头文件后缀：.h，执行文件后缀：.cpp。文件名应和文件中定义的主要或者唯一类的名字相同

例如：class GBTerrainMgr，包含此类的文件名应为：gbterrainmgr.h和gbterrainmgr.cpp

## 类命名与结构体命名

首字母大写，应使简明正确的英文单词或者缩写组成名称，并在名称前加“GB”表示本项目相关。

例如：class GBMesh，表示GB项目，Mesh表示该类负责处理MESH有关的操作

## 常量与宏定义

常量名称和宏定义的名称应采用全部大写，由表示含义的单词或者缩写表示，单词与单词之间用“\_”连接，并前缀添加“GB”表示本项目相关。

例如：enum GB\_ENTITY\_TYPE，表示实体的类型

## 命名空间的名称命名

命名空间应采用首字母大写，简洁的单词表示名称

例如：namespace GBClient，表示客户端部分代码

## 函数命名

使用小写字母开头，由多个单词或者单词缩写组成

例如：int getVertexCount()，表示返回顶点个数

## 变量命名

类中的成员以“m”开头，后接多个单词或者单词缩写组合

例如：int mVertexCount，表示顶点数量

全局变量以“g”开头，后接多个单词或者单词缩写组合

例如：char\* gNameStr，表示名字的字符串

局部变量和参数的命名，应该以类型的简写或者可标示变量类型的字母开头，然后使用尽量简洁的单词组成名称

例如：char\* szTmp，表示临时字符串

# 代码书写规范

## 头文件使用规范

### 头文件应避免重复包含，重复编译，使用#ifndef,#define,#endif的结构解决此问题，对不涉及到实现细节的类的引用应该采用声明的方式而不是包含它的头文件

例如：

#ifndef IDENTIFIER\_H

#define IDENTIFIER\_H

// 将使用GBThread的声明

class GBThread;

class GBThreadMgr

{

// 类中其它部分

private:

std::list<GBThread\*> mThreadList; // 表示所拥有的线程链表

};

#endif // IDENTIFIER\_H

### 头文件应尽量保证只有一个类，如果必须要包含两个以上类，应该以最主要的类的名称命名文件名

## 代码流程规范

### 代码流程应简单易懂，函数调用应做到适当，函数不宜过大，完成单元功能即可

### 分支条件判断，使用switch语句时必须有default分支，使用if语句时必须有else分支

例如：

switch(tmpClr)

{

case GREEN:

// …

case RED:

// …

default:

// …

}

### 不要使用goto语句进入另外分支，会影响可读性

### 函数必须保证每条分支都具有返回值，并且在函数的结尾必须使用return返回，void函数除外

例如：

bool isAEqualB(int a,int b)

{

if(a == b)

return true;

else

return false;

retrun false;

}

## 内存及指针规范

### 在同一作用域中new和delete

例如：

void swapBuffer(Class\* a,Class\* b)

{

if(a!=0 && b!=0)

return;

// tmp在函数内部申请内存，应该在内部释放

Class\* tmp = new Class();

tmp->Clone(a);

a->Clone(b);

b->Clone(tmp);

delete tmp;

tmp = 0;

}

void callFunc()

{

// a和b在swapBuffer外部申请，外部释放

Class\* a = new Class();

Class\* b = new Class();

swapBuffer(a,b);

delete a;

a = 0;

delete b;

b = 0;

}

### 应尽量避免指针运算

### 在必要时应该用const修饰指针，比如传递的参数，函数返回的类级别指针

例如：

void setA(const A\* pA)

{

// …

}

### 指针的空值应使用数值0，而不是定义成NULL

### 所有指针在使用之前必须通过检测，并在释放之后必须设置成0

### 应该使用new和delete处理类的分配和释放，只有在必要的时候才能使用malloc和free

## 变量及函数的定义与使用

### 多个变量应分开定义，并且进行初始化

例如：

int a = 5;

int b = 10;

### 基本类型需保证一致的使用，比如统一使用BOOL而不是bool，这个BOOL应该是统一定义的

例如：typedef int BOOL;

### 复杂的类型，或者自定义的类，应该注意使用引用和指针传递，可以提高效率

例如：

void setA(const A& a)

{

// …

}

### 简洁的函数应该显示表示inline关键字，inline函数不应该过于臃肿

例如：

inline void setA(int a)

{

mA = a;

}

### 应使用C++定义的函数和类型，尽量避免采用C的定义

例如：使用iostream中的方法，而不是stdio中的方法

### 变量类型的转换应使用显示的转换语句，并且尽量使用C++的转换语法，在使用中应尽量减少转换，以提高安全性

例如：使用dynamic\_cast和static\_cast，而不是直接使用(Type)这种方法转换

### 不要把常量转换成非常量

## 面向对象的程序规范

### 必须显示的定义一个无参的构造函数

### 对于有虚函数的继承体系，必须定义virtual析构函数，copy构造构造函数

### 对于有虚函数的继承体系，如果必要，应定义赋值操作

### 对于需要子类重写的虚函数必须在父类和子类显示的标示virtual关键字

### 应当尽量使用public继承，并且使用protected和private隐藏类级别成员

### 构造函数中必须对应该初始化的成员进行初始化，子类的初始化应按照构造顺序进行书写

例如：

Class Base

{

public:

Base() : mA(5){}

// 其它内容

private:

int mA;

};

Class Derive : public Base

{

public:

Derive() : CBase(), mB(10){}

// 其它内容

private:

int mB;

};

### 应尽量避免使用友元

### 模板的instantiation操作应该由程序员明确要求

template class 类名称<类型名称>;

### 应尽量避免以一个父类指针指向一个子类对象数组，如果必须这么做，请注意在new和delete时变成子类指针进行处理

请不要使用类似语句：Base\* b = new Derive[n]，n表示数组元素个数，这会在内存分配和释放时产生问题

### 在继承体系中不应该使用placement new operator创建对象

### 除非必要，不要使用虚拟继承和多继承，在多继承中应采用类似JAVA接口的方式继承

## 其它代码书写规范

### 域中的代码必须使用{}包含，{}应分开两行书写，并且每一行只能书写一条语句

例如：

// 这里即使getA只有一条语句，也应该从另一行写起，这是为了方便断点调试

int getA()

{

return mA;

}

### “.”，“->”，“()”“{}”之前不应该有空格

### 在作用域外部时需显示写明函数，变量的作用域

例如：

namespace A

{

void func()

{

// …

}

}

namespace B

{

void callFunc()

{

// 这里即使前面有using namespace A，也应该写清A::

A::func();

}

}

### 若是非明确执行顺序的语句，就应该使用括号

例如：

if((a=b) && (c+d)){…}

## 注释规范

### 使用“//”开头加入注释

### 注释应简明，表达清楚用意即可

### “#endif”后面应该有相应的“#if”的注释

### 需要临时删除的代码可以采用“/\*\*/”的方法注释掉

# 代码编译部署规范

## 每个项目建立在devwork下面的文件夹的子文件夹里

## 本项目所有解决方案放在devwork下面的文件夹的根目录中

## 编译的中间文件放在devwork\intermediate\$(ConfigurationName)\$(ProjectName)

## 编译的依赖库放在devwork\lib下面

## 编译的可执行文件放在devwork\bin下面

## 一切调试版本的生成文件后面需加上[d]以示区分

## 所有生成文件不得上传SVN库

## 项目设置中的所有引用路径都要使用相对路径