GamePlay: C++ Coding Guidelines

Files

Files should be named according to their class names:

class Foo = ***Terrain****.h* + ***Terrain****.cpp*

Naming

|  |  |
| --- | --- |
| Classes, Structs, Enums and Typedefs | CamelCase |
| Functions | camelCase |
| Private + Protected member variables | \_camelCase |
| Public member variables | camelCase |
| Local variables | camelCase |
| Constants | ALL\_CAPS |
| Enum values | TYPE\_ALL\_CAPS |

Class Layout

#pragma once

#include "Vector3.h"

namespace gameplay

{

class Foo

{

public:

Foo();

~Foo();

Vector3 getSimple() const;

void setSimple(const Vector3& simple);

private:

Vector3 \_simple;

};

}

Classes

* Each access modifier appears no more than once in a class, in the order:

public, protected, private.

* All public member variables are at start of the class.
* All private member variables live at the end of the class.
* Use protected member variables judiciously.
* Constructors and destructors are the first methods in a class after public member variables.
* The method impl. in cpp appears in the order which they are declared in the class.
* Avoid inline implementations absolutely needed and trivial and short.

Enums

* Should be declared in classes directly after constructor and destructors.
* Keep the naming simple and short-and-sweet.
* Enum values should be prefixed with the enum type name:

public:

Camera();

~Camera();

enum Mode

{

MODE\_PERSPECTIVE,

MODE\_ORTHOGRAPHIC

};

Camera::Mode getMode() const;

# Smart Pointers vs Pointers

* Use std::unique\_ptr<Foo> whenever possible especially for public api pointers.
* Any code still using normal pointers should be slowly be migrated to std::unique\_ptr.
* Any delete function call appearing in the code is a red flag and needs a good reason.
* Use std::shared\_ptr<Foo> only when sharing is required.
* Use std::vector if you need an array and use std::vector::data() to get a raw pointer to the beginning of a vector.

# Auto

* Use auto sparingly and when needed to abstract and complex type.
* Great for use on std:: STL containers and smart pointers.
* Be careful about accidental *copy-by-value* when you meant *copy-by-reference*.
* Understand the difference between auto and auto&.

# Lambdas

* Lambdas are acceptable “where they make sense”.
* Ensure use is focused around anonymity.
* Avoid overuse, but especially for std algorithms (std::sort, etc) they are perfectly fine.

# Range-based loops

* They’re great, use them.
* They don’t have to be combined with auto
* They are often more readable to not use auto for simple types:

for (int dev : devices) - More obvious that the iterator is a device index than.

for (const auto& dev : devices) - Suggests ‘dev’ might be of type Device.

# Friend

* Avoid using friend unless needed to really restrict access to internal classes.
* It easily leads to difficult-to-untangle interdependencies that are hard to maintain.
* We will continue to remove where possible in existing code.

# size\_t

* Use size\_t for all counts and accessing std::size results.

# int

* We use int for dimensions, lengths and integer values.

# unsigned int

* Avoid using unsigned int unless needed for bitwise flags, masks, etc.

# float

* The float type is the primary precision for all data in the 2D and 3D world.
* Used for low precision time intervals.
* Use double is only used when necessary for higher precision.

# nullptr

* Use nullptr instead of *0* or *NULL*.

# Errors and Logging

* Use GP\_ERROR macro for all errors that will stop the program in release and debug modes.

GP\_ERROR("Invalid json base64 string for propertyName: %s", propertyName);

* Use GP\_INFO for one time logging events such as initialization or unexpected singularities.
* Avoid excessive logging which can impact game engine performance. Logging every frame.

# GP\_ Macros and Global Constants

* Use existing GP\_XXX for various functionality.
* Use GP\_ as a prefix for all gameplay scoped macros and global constants.

# Assertions

* Use GP\_ASSERT for quick danger checks in start of impl that’s checked for in debug mode.

size\_t SerializerJson::readFloatArray(const char\* propertyName, float\*\* data)

{

GP\_ASSERT(propertyName);

GP\_ASSERT(\_type == Serializer::TYPE\_READER);

Public member variable access

* Use this-> when accessing public variables.

void BoundingSphere::set(const Vector3& center, float radius)

{

this->center = center;

this->radius = radius;

Public member variable initialization

* Initialize public member variables directly in .h file for awareness.

class Rectangle

{

public:

float x = 0.0f;

Protected + private member variable initialization

* Initialize all protected and private member variables in constructor in cpp to obscure.
* Initialize all protected and private member variables in the order they are declared.

Game::Game() :

\_config(nullptr),

\_state(Game::STATE\_UNINITIALIZED),

\_width(GP\_GRAPHICS\_WIDTH),

# Header Comments

* Avoid spelling and grammatical errors.
* Header comments use *doxygen format*. We are not too sticky on *doxygen* formatting policy.
* All public functions and variables must be documented.
* The level of detail for the comment is based on the complexity for the api.
* Most important is that comments are simple and have clarity on how to use the api.
* @brief is dropped and automatic assumed on first line of code. Easier to read too.
* @details is dropped and automatic assumed proceeding the brief line.
* @param and @return are followed with a space after summary(brief) or details.

*Example:*

/\*\*

\* Tests whether this bounding box intersects the specified bounding box.

\*

\* You would add any specific details that may be needed here. This is

\* only necessary if there is complexity to the user of the function.

\*

\* @param box The bounding box to test intersection with.

\* @return true if the specified bounding box intersects this bounding box;

\* false otherwise.

\*/

bool intersects(const BoundingBox& box) const;

* Overridden functions can simply refer to the base class comments

/\*\*

\* @see Serializable::onSerialize

\*/

void onSerialize(Serializer\* serializer);

# Implementation Comments

* Clean simple code is the best form of commenting.
* Do not add comments above function definitions in .cpp they are already in header.
* Used to comment necessary non-obvious implementation details not the api.
* Only use // line comments on the line above the code you plan to comment.
* Avoid /\* \*/ block comments. Preventing others from ‘easily’ doing their own block comments when testing, debugging, etc.
* Avoid explicitly referring to identifiers in comments, since that’s an easy way to make your comment outdated when an identifier is renamed.

Formatting

* You should set your IDE or source code editor to follow the formatting guidelines.
* Keep all code less than ***120*** characters per line.

Includes

* #pragma once is first line of the .h class header.
* #include “Base.h” is the first line of the .cpp class implementation.
* Include the corresponding class header to the second line just after including “Base.h”.
* All the other headers, sorted by “local to distant” directories. (Ex. std headers come last)

# Tabs

* Insert **4** spaces for tabs.
* Change your ide or editor to replace tabs for spaces.
* Avoid tabs wars.

# Line Spacing

* One line between gameplay namespace.
* One line of space between functions declarations in source and header.
* One line after each class scope section in header.
* Function call spacing:
  + No space before bracket.
  + No space just inside brackets.
  + One space after each comma separating parameters.

serializer->writeFloat("range", \_range, LIGHT\_RANGE);

* Conditional statement spacing:
  + One space after conditional keywords.
  + No space just inside the brackets.
  + One space separating commas, colons and condition comparison operators.

if (enumName.compare("gameplay::Light::Type") == 0)

{

switch (value)

{

case Light::TYPE\_DIRECTIONAL:

return "TYPE\_DIRECTIONAL";

...

* Do not attempt to align blocks of variables to match spacing. This just cause unnecessary changes in code with variables are added in between.
* Align indentation space for parameters when wrapping lines to match the initial bracket.

*Examples:*

Matrix::Matrix(float m11, float m12, float m13, float m14,

float m21, float m22, float m23, float m24,

float m31, float m32, float m33, float m34,

float m41, float m42, float m43, float m44)

return sqrt((point.x - sphere.center.x) \* (point.x - sphere.center.x) +

(point.y - sphere.center.y) \* (point.y - sphere.center.x) +

(point.z - sphere.center.z) \* (point.z - sphere.center.x));

* Use a line of space within .cpp implementation function to help organize blocks of code.

// Lookup device surface extensions

...

...

// Create the platform surface connection

...

...

...

# Indentation

* Indent next line after all braces { }.
* Move code after braces { } to the next line.
* Always ident the next line of any condition statement line.

*Examples:*

if (box.isEmpty())

return;

for (size\_t i = 0; i < count; ++i)

{

if (distance(sphere, points[i]) > sphere.radius)

{

return false;

}

}

* Never leave conditional code statement on same line as condition checks:

if (box.isEmpty()) return;