# Introduction:

## Purpose:

This software document describes the architecture and design of google’s game engine——liquidfun, which is mostly based on another open-source physical stimulating engine,Box2D. People that are interested in making 2D games with functions such as 2d rigid-body, fluid stimulation and so on with liquidfun may find this document useful.

## Scope

Liquidfun provides support for procedural animation of physical bodies to make objecys move and interact in realistic ways. The whole project can be divided into two parts: one is Box2D, the other is liquidfun codes added. The Box2D contains three main folders: Collision, Common and Dynamics. There are 45 headerfiles and 45 source files in the main part of box2d. And liquidfun adds a new folders: particles. This includes 6 headfiles and 5 source files. In total, there are near 30015 lines(comments and blank lines included) in the project. The whole project is used to stimulate how liquid moves with fluid particles.

## Overview

Our document mainly aims at explaining the system architecture, the data design, the component design, GUI design, oop features and what we have learned in the process of code reading.

## Definitions and acronyms

Here list meanings of some terms and abbreviations in the project:

b2Vec(n): a n-dimension vector

b2AABB: a rectangle that all edges parallels X axis or Y axis

b2Dot: dot product of two vectors

b2Cross: cross product of two vectors

Manifold: the zone two shapes collide

I: Moment of Inertia

TOI:Time of impact

## 1.5 Basic OOP Methods

Box2D mainly uses many classes with different functions to build the system. Some classes are used to imitate physical objects, others may be used to calculate some physical attributes. Also, to implement shapes, polymorphism and fundamental factory design patterns are used. Detailed OOP methods are recorded in following docs files.

# System Overview

Liquidfun is a physical engine based on Newton's physical law. Through continuous scanning and integral operation, Liquidfun can calculate the position and velocity of different objects at every timestep. Also, Liquidfun can simulate various relationships and state data between objects such as friction and collision. It will calculate the force and moment of the object iteratively in a period of time to meet these needs(Supporting simultaneous detection of multiple objects）. And users can also change the mass、position、shape、friction、density and so on to satisfy their own needs.

# System Architecture

### 1.Architecture Design

Liquidfun can be mainly divided into four parts. Collison part handles how to create objects with different shapes. It also handles collision between two rigidbodys and calculates the raycast. Dynamic part handles how to create the joints and divides objects into different “islands” for easier processing. It also creates the physical world to control every timestep. Particles is added to stimulate fluid. Common part defines many basic funtions. Most mathematic calculations is defined in this part. It also creates some allocators to allocate memory and store data. Settings and many definitions is also included in it.

In Liquidfun there are some basic types:

Body: Store some basic information about objects

Fixture: Set some data of an object (such as mass、shape、density etc.)

Joint: Set some constraints between specified objects

Contact: Setting the interaction relations between different objects

And beyond these, Liquidfun can simulate some situations in reality.

Collision: In this section, Liquidfun created many kinds of shapes using a base class b2shape. And through BroadPhase and DynamicTree (and many useful algorithm such as GJK algorithm and Gift Wrapping algorithm), Liquidfun can quickly calculate the positions and velocities of many objects and determine whether two objects have (or will) collided. And finally simulate collision between objects that created before (such as collision between edge and polygon, circle and polygon and so on).

Dynamic Module is the key part to support the initialization of a new game. Dynamic module is based on the common and collision module and is associated with user interface. Common, collision and particle modules provide the basic internal classes and functions, such as b2DynamicTree, b2BroadPhase and others for mathematic calculation, while dynamic module contains classes and functions which make up the user interface and undertake the real-time computing tasks. When user set up a new game, a virtual world will be constructed through b2World’ constructor in dynamic module. In the world, new bodies and their attached fixture, contacts and joints are built through their corresponding classes’ constructors and “Create” function which is also in dynamic module. Force and torque can be applied to the bodies through contacts and joints, and the bodies gain velocity and change position which stored in b2Body class. The position and velocity change continuously, sometimes collision happens, so real-time computing process is conducted by b2World’s function, “Solve” and “SolveTOI”, and the internal classes such as b2Island. Time is divided into many small time-steps (1/60 second for each time-step by default), which is convenient for real-time computing.

Particle, along with particle system, are new-added classes in liquidfun. Liquidfun uses particles to stimulate the environment. By using functions provided by box2d and their own functions, liquidfun can process many physical events. Particles are divided into many different kinds and each kind of particles have different functions. Also, particles can be paired or tripled. The particle system can store particles and their data. It can also update the data stored in the buffers by those solve functions. It can use these data to handle contacts of particles and destructions of particles and particle groups. Notice that the b2world file is edited in liquidfun, because the world should also be able to handle particles’ actions.

Controllers: This section is newly contributed in liquidfun, too. In liquidfun, users can use controllers to conveniently control bodies’ movement in the certain environment (world). In general, Controllers are a convenience for encapsulating common per-step functionality. There is a abstract class named b2Controller to create other concrete Controllers. Specifically, Controllers are divided in 6 concrete controllers, such as b2GravityController for control gravity’s effect on bodies connected. In addition, the class named b2ControllerEdge need to be emphasized for it record the connected bodies’ list for the certain controllers and the connected controller’s list for the certain bodies.

### 2.UML

UML for most classes of liquidfun is in UML folders.

# Data Design

General Introduction: Allocators are used to allocate memory for different objects. There are in all three kinds of concrete allocators for allocations of blocks, stack and slab. Details are as follows.

1. b2BlockAllocator:
2. Path: Box2D/Common/b2BlockAllocator.h

Box2D/Common/b2BlockAllocator.cpp

1. Headerfile:

Box2D/Common/b2Settings.h

Box2D/Common/b2TrackedBlock.h:

1. Introduction:

This is a small object allocator used for allocating small objects that persist for more than one time step.(block)

1. Const:( b2BlockAllocator.h中定义的一些常数)

const int32 b2\_chunkSize = 16 \* 1024

const int32 b2\_maxBlockSize = 640

const int32 b2\_blockSizes = 14

const int32 b2\_chunkArrayIncrement = 128

1. Struct:

A)b2Chunk:

int32 blocksize

b2Block \* next

B)b2Block: This struct is used to store the address of blocks .

b2Block \* next

1. Class:

b2BlockAllocator

1. Data members:

Private:

m\_chunks(b2Chunk\*, b2Chunk is a struct defined in b2BlockAllocator.cpp)

m\_chunkCount(int32)

m\_ chunkSpace(int32)

s\_blockSizes[b2\_blockSizes==14](**static** int32型数组)：

m\_giants(b2TrackedBlockAllocator): Record giant allocations, the ones bigger than the max block size

s\_blockSizeLookup[b2\_maxBlockSize+1== 641]( **static** uint8型数组)：

s\_blockSizeLookupInitialized（**static** bool）

B) Member function:

Public:

**b2BlockAllocator**():

Constructor

**~b2BlockAllocator**():

Distructor

void\* **Allocate**(int32 size):

Allocate memory for one block. The function will use b2Alloc if the size is larger than b2\_maxBlockSize

void **Free**(void\* p, int32 size)：

Free memory for one block. The function will use b2Free if the size is larger than b2\_maxBlockSize.

void **Clear**()：

Free all memories allocated for blocks.

uint32 **GetNumGiantAllocations**() const:

Returns the number of allocations larger than the max block size.

1. b2StackAllocator
2. Path:

Box2D/Common/ b2StackAllocator.h

Box2D/Common/ b2StackAllocator.cpp

1. Headerfile:

Box2D/Common/b2Settings.h

Box2D/Common/b2Math.h

string.h

1. Introduction:

This is a stack allocator used for fast per step allocations. You must nest(嵌套) allocate/free pairs. The code will assert（强行插入） if you try to interleave multiple allocate/free pairs.

1. Const:

const int32 b2\_stackSize = 100 \* 1024

const int32 b2\_maxStackEntries = 32

1. Struct:

b2StackEntry: This struct is used to record the info of stacks.

char\* data

int32 size

bool usedMalloc

1. Class:

b2StackAllocator

1. Data members:

Public:

enum { MIN\_ALIGNMENT = sizeof(void\*) }（必须为2的指数倍）

enum { ALIGN\_MASK = MIN\_ALIGNMENT - 1 }

Private:

m\_data[b2\_stackSize](char型数组，字符串)

m\_index(int32)

m\_allocation(int32)

m\_maxAllocation(int32)

m\_entryCount(int32)

m\_entries[b2\_maxStackEntries==32]( b2StackEntry型数组)

1. Member function:

Public:

b2StackAllocator()：  
 Constructor.

~b2StackAllocator():

Distructor.

void\* Allocate(int32 size):

Allocate memory for one stack

void\* Reallocate(void\* p, int32 size):

Reallocate memory for one stack

void Free(void\* p):

Free memory for one stack.

int32 GetMaxAllocation() const:

Return the max allocation for the stack.

1. b2SlabAllocator
2. Path:

Box2D/Common/ b2SlabAllocator.h

1. Headerfile:

stddef.h

stdint.h

new

Box2D/Common/b2IntrusiveList.h

Box2D/Common/b2FreeList.h

Box2D/Common/b2Settings.h

Box2D/Common/b2TrackedBlock.h

1. Introduction:

This allocator is the freelist based allocator for fixed sized items from **slabs** (memory preallocated （预分配）from the heap).

All objects in a **slab** are constructed when a slab is created and destructed when a slab is freed.

1. Template:

template<typename T>:

T should be a class which has a default constructor and implements the member function "b2IntrusiveList\* GetListNode()".

1. Class:
2. Slab:
3. Data member:

Private:

m\_numberOfItems(uint32)

m\_padding[b2\_mallocAlignment-sizeof(uint32)](uint8型数组)

1. Member function:

Public:

**Slab**(uint32 numberOfItems):

Constructor to initialize a slab with the number of items it contains.

**~Slab**()：

Distructor.

uint32 **GetNumberOfItems**() const:

Get the number of items in this slab.

T\* **GetFirstItem**() const:

Get a pointer to the first item in the slab.

T\* **GetItemEnd**() const:

Get a pointer to the end of the slab.(*This is a pointer after the last byte of the slab but not the last item in the slab.*)

1. b2SlabAllocator:
2. Data member:

Private:

class Slab:

The class Slab is defined in class b2SlabAllocator.

m\_slabs(b2TrackedBlockAllocator):

Contains a list of b2TrackedBlock instances where each b2TrackedBlock' s associated user memory contains a Slab followed by instances of T.

包含b2trackedBlock实例的list，其中每个b2trackedBlock的**关联用户内存**包含一个slab，后跟T的实例。

m\_itemsPerSlab（uint32）:

Number of items to allocate in the next allocated slab.

b2TypedFreeList<T> m\_freeList:

Freelist which contains instances of T.

1. Member function:

Public:

**b2SlabAllocator**(const uint32 itemsPerSlab):

Constructor to initialize the allocator to allocate itemsPerSlab of type T for each slab that is allocated.

**~b2SlabAllocator**():

Distructor.

void **SetItemsPerSlab**(uint32 itemsPerSlab):

Set size of the next allocated slab using the number of items per slab. Setting this value to zero disables further slab allocation.

将每个slab的项目数设置为下一个被分配内存的slab的大小。将此值设置为零将禁用下一步的slab内存分配。

uint32 **GetItemsPerSlab**() const：

Get the size of the next allocated slab.

T\* Allocate()：

Allocate an item from the slab.

void Free(T \*object):

Free an item from the slab.

bool AllocateSlab():

Allocate a slab, construct instances of T and add them to the free pool.

分配一个slab，构造T的实例并将它们添加到自由池。

void FreeAllSlabs():

Free all slabs.

void FreeEmptySlabs():

Free all empty slabs.

(This method is slow - O(M^N) - since this class doesn't track the association between each item and slab.)

const b2TypedFreeList<T>& GetFreeList() const:

Get the item allocator freelist.

Private:

void FreeSlab(Slab \* const slab):

Destroy all objects in a slab and free the slab.

Slab\* BlockGetSlab(void \*memory):

Get a pointer to a Slab from a block of memory in m\_slabs.

T\* SlabGetFirstItem(Slab\* slab):

Get a pointer to the first item in the array of items referenced by a slab.

# Component Design

We have listed the use of all main functions in the following docs.

# Human Interface Design

Please see following docs “user interface”.

# Design Patterns And Polymorphism Features

Polymorphism:

b2Collision.cpp: bool b2TestOverlap(const b2Shape\* shapeA, int32 indexA, const b2Shape\* b2shapeB, int32 indexB, const b2Transform& xfA, const b2Transform& xfB) Judging whether two shapes coincide.

b2Distance.cpp: void b2DistanceProxy::Set(const b2Shape\* shape, int32 index) First, it will get the accurate type of the shape and using mandatory type conversion (static\_cast) to make shape become one of b2Shape’s derived class.

b2Shape:

b2Shape\* b2Shape::Clone(b2BlockAllocator\* allocator) const :Copy a given shape.

virtual int32 GetChildCount() const:get the number of child primitives

virtual bool TestPoint(const b2Transform& xf, const b2Vec2& p) const:test whether a

point is in the shape

virtual void ComputeDistance(const b2Transform& xf, const b2Vec2& p, float32\* distance, b2Vec2\* normal, int32 childIndex) const:for convex, calculate the distance between it and a given point.

virtual bool RayCast(b2RayCastOutput\* output, const b2RayCastInput& input,const b2Transform& transform, int32 childIndex) const: to cast a ray on the derived shape.

virtual void ComputeAABB(b2AABB\* aabb, const b2Transform& xf, int32 childIndex) const:

compute the AABB that can contain the shape.

virtual void ComputeMass(b2MassData\* massData, float32 density) const:calculate the mass of the given shape using dimensions and density.

# Code Skills We Have Learned

1. override functions can use other versions of this function.
2. In b2Body, b2Contact, etc., int32 flags are often used, which takes full advantage of every bits to store the related information and is convenient to use for checking and judging the cases (like bool)
3. AllowSleep(if a body doesn’t move for a long time, it turns into a sleep mode) skills avoid unnecessary calculation of the irrelated bodies during the collision, which save memories.
4. WarmStarting skills save the current message of a moving body in one time-step and reuse the message after the next time step starts. This allows the bodies to move continuously without consuming too much memories.
5. Island class is specially set to undertake the major part of computing during each time step and destroyed after used. This can help better management of the memory and maintain stable computing process.
6. “def” is often used as a single parameter when creating a complex class. “def” contains several data members to initialize the class, which makes it easier and more accurate to initialize the class’s data members. They are very useful in Particles’ class.
7. Proxy pattern is used to replace the parent body and its attached fixture on the broadPhase. A proxy class hold fewer data than a whole b2Body and b2Fixture class so that it has higher efficiency being computed on the broadPhase.
8. The class (b2Fixture, b2Contact, etc.) separated the create functions from the constructor function because the destructor cannot access the allocator.
9. b2Contact class use function pointer instead of polymorphic programming to deal with the problem: different derived class from b2Contact has different “Create” and “Destroy” function. In the base class b2Contact, there is a static member array “s\_registers” storing the derived functions. So when calling the derived class’s “Create” or “Destroy” functions, the base class’s “Create” or “Destroy” function will be called first, and the base class’s function calls the derived class’s function (“Create” or “Destroy”).
10. use some mathematic skills to compute 1/x^0.5, which saves time.
11. Use typedef to define function pointers for easier use.