Timeline Manual

Chen Sheng(陈胜)

# Introduction

Timeline is my innovation and solution for asynchronous programming. It has been extensively used by every mobile game our department developed, thus has proved to be reliable. Its first use was introduced in SD1, where almost every asynchronous progress in game play was implemented as a Timeline and organized hierarchically to make the whole game run. Later, it was harnessed in Bingo project to create some of the most complicated animations in the game and to deal with login process. In Deck Heroes, it’s utilized to implement combat system, login process, resources downloading and 2 sub modals.

## 1.1 The concept of Timeline

“Give me an update function; I can create a whole game.” The update function plays a central role in game programming. With a flexible mechanism to well organize update functions, we are able to divide major problems into minor ones and solve them separately. To do that, we need a model to represent any asynchronous progress and to compose them into a tree structure. We have seen people doing these for actions in all kinds of framework to support animations. To step furthermore, we can extend this model to every process in a game, which leads to the concept of Timeline. We can see in practice how this allows us to better reuse codes and to integrate different functions in a loose coupling manner.

# Structure

To serve the needs that give birth to the concept of Timeline, the structure designed should be versatile enough to model the asynchronous problems that arise in game programming. When Timeline was designed, I had 4 goals in mind:

1. A Timeline should be able to do its own business and support sub or child Timelines.
2. Sub Timelines can be added to or removed from a parent Timeline dynamically and easily.
3. The parent Timeline should allow sub Timelines running both concurrently or in a sequence.
4. A Timeline should allow the strict control of its life cycle.

To achieve goal 1), every timeline should be able to play both the role of leaf and branch in a tree structure. This is important and distinguishes Timeline from many previous designs. We do not need a specific container class to compose sub Timelines, because whether a node is a leaf or branch in a Timeline tree would subject to changes in design and implementation very sensitively. And goal 3) should be achieved for the same reason, whether sub Timelines should be running concurrently or in a sequence is also very likely to change. We do not want to change the base class or add an extra container Timeline just for reasons like 2 animations running concurrently should be modified to be running in a sequence. What’s more, for a Timeline to represent a real game, it is likely to have sub Timelines running both concurrently and in a sequence. So if the base Timeline class is able to support its own business, serve as the container of sub Timelines and allow running of sub Timelines both concurrently and in a sequence, we can achieve the flexibility we desire for the aforementioned reasons. And we will explain later in this article why the base Timeline class should be designed in this way to achieve goal 2) as well. We are going to demonstrate this with a diagram.

Internal business

Sub Timeline A

Sub Timeline B

Sub Timeline C

Sub Timeline D

Sub Timeline E

Timeline P

5s

2s

Fig.1 A diagram of Timeline structure

In Fig.1, Timeline P has its own business, and 5 sub Timelines, A, B, C, D, and E. As illustrated by the diagram, A and B are supposed to running concurrently with A being delayed by 5s. C and D are supposed to start running after A is done. And E is proceeded by D, and will start running in 2s when D ends.

We can represent the relations in the diagram with the following codes. Forgiving me for violating coding conventions here, I have to use simple variable names in the diagram.

P->addSubTimline(A, NULL, 5.0f);

P->addSubTimeline(B);

P->addSubTimeline(C, A);

P->addSubTimeline(D, A);

P->addSubTimeline(E, D, 2.0f);

In this way, we can easily compose the above Timelines without introducing any extra container objects. If we use the composite model of IAction in Ui library, the equivalent code would be something like the following.

ActionSequence\* pDE = new ActionSequence();

pDE ->addAction(D);

pDE ->addAction(new ActionDelay(2.0f));

pDE ->addAction(E);

ActionCombine\* pCDE = new ActionCombine();

pCDE->addAction(C);

pCDE->addAction(pDE);

ActionSequence\* pACDE = new ActionSequence();

pACDE->addAction(new ActionDelay(5.0f));

pACDE->addAction(A);

pACDE->addAction(pCDE);

ActionCombine\* pABCDE = new ActionCombine();

pABCDE->addAction(pACDE);

pABCDE->addAction(B);

ActonCombine\* pABCDEP = new ActionCombine();

pABCDEP->addAction(P);

pABCDEP->addAction(pABCDE);

Compared with the Timeline counterpart, the IAction version requires more code. However, what is worse is that it introduces many unnecessary objects, and makes the main structure hard to understand. If we want to dynamically remove a sub object from the tree, we would have to track the container objects like pABCDEP, pDE, pABCDE, pCDE to know where to remove the sub objects from. Now imagine a scenario that a change in the game requires us to make C running after B, and D and E running concurrently after A is done. We can simply modify our Timeline version to the following.

P->addSubTimline(A, NULL, 5.0f);

P->addSubTimeline(B);

P->addSubTimeline(C, B);

P->addSubTimeline(D, A);

P->addSubTimeline(E, A, 2.0f);

Now, take your time to figure out how many changes you should make to codes of IAction versions. You will find out that the old container objects (pDE, pCDE and pACDE) introduced for composing the objects are no longer needed. And you will have to create new container objects (pBC, pDE and pADE) to reflect the new relations of the objects. Now imagine that you need to dynamically remove A, B, C, D or E from the tree. Does it seem like a disaster to you? That’s why I have stated before in this article that to achieve goal 2) we also require Timeline to be designed in this way. Without introducing any extra intermediate object, we can simply remove the sub Timelines with an expression like P->removeSubTimeline(A).

You may have figured out why we should not introduce extra container objects just for the reasons making some sub objects in a sequence or in currency. Because in reality, there are no such objects, they reflect time sequence/concurrency relation other than logical relation which is more stable than the former one. This leads to a design principle of Timeline, only creating “natural objects” not the “artificial ones”.

# Life Cycle

Understanding the Timeline life cycle is crucial for using Timeline. The following is a diagram describing the main phases of Timeline life cycle.

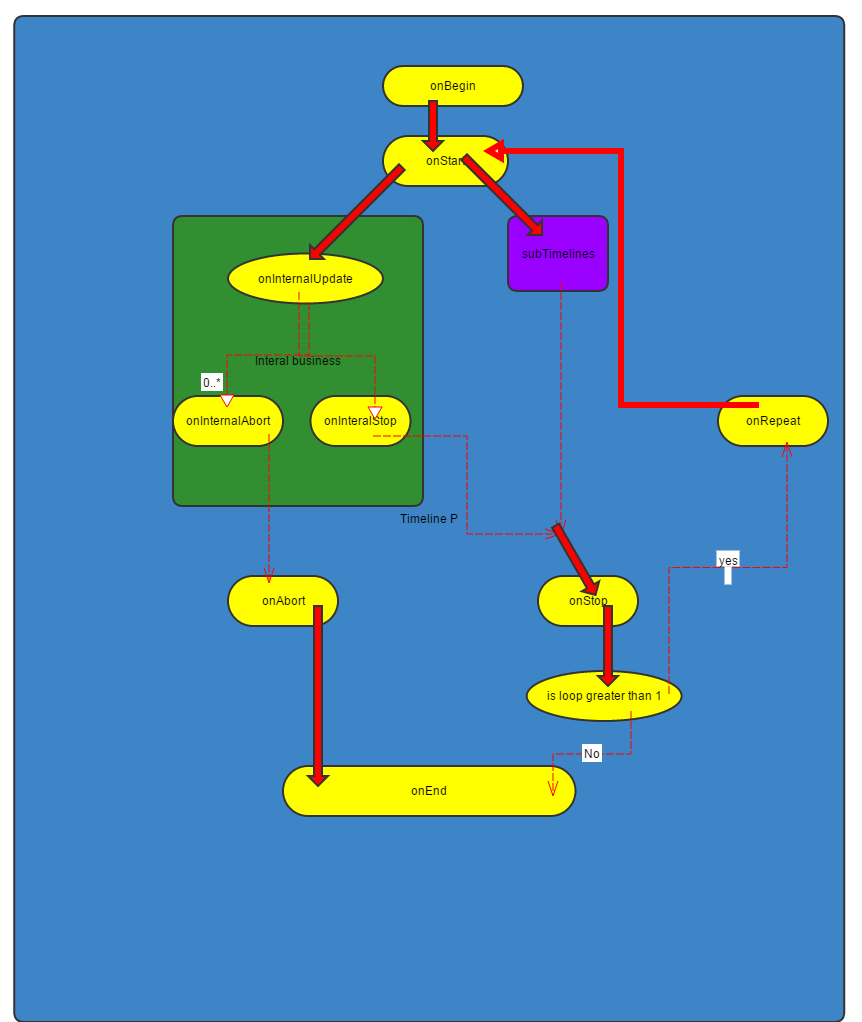


Fig.2 Timeline life cycle

Timeline supports looping; the start of the whole Timeline is different from that of a single loop. When the whole Timeline begins, the virtual function onBegin will be called, while a single loop starts, the virtual function onStart will be called. So if you want to do any initialization that happens only once in the life cycle of a Timeline, you can override onBegin for that. And for tasks that need to be done at the start of every single loop, onStart is the best place. When a Timeline starts, its update function will be called, which will call virtual function onInternalUpdate to do its own business and call the update function of its sub objects to update their business. If onInternalUpdate returns false, it’s considered that its own business is done, so the virtual function onInternalStop will be called. And if the internal business and all unskippable sub Timelines are finished, the single loop of the Timeline is considered finished, and the virtual function onStop will be called to signify the end of a single loop of the Timeline. At this time, if the current loop is less than total loop or the total loop is infinite(-1), the Timeline will be repeated with virtual function onRepeat being called, then a new loop starts with onStart being called again and the sub Timelines that are still attached being restarted. Otherwise the current loop is greater or equal to the total loop, the virtual function onEnd will be called, then the whole Timeline is considered to be done. This is the normal process, a Timeline can be forced to be ended when at least its internal business or one sub Timeline has not finished. If this happens, the Timeline is considered to be aborted. If the internal business hasn’t been done yet when the Timeline is aborted, the virtual function onInternalAbort will be called. After that the virtual function onAbort will be called. With different functions being called, it allows us to deal with the exceptional abort separately from the normal stop. Within the call of Timeline’s update function, after its own internal business and sub Timelines being updated, a virtual function named onPostUpdate will be called, providing us an opportunity to do something afterward. On each phase of the life cycle, besides of calling the virtual functions to allow sub classes customize their own behaviors, a TimelineEvent is dispatched to notify the outside its state. For example TIMELINE\_BEGIN will be dispatched after onBegin is called, and TIMELINE\_ABORT will be dispatched after onAbort is called.

So from the diagram we can see that, onBegin will be matched by onEnd, while onStart will be matched by either onInternalAbort or onInternalStop and either onAbort or onStop. From this, we can conclude that if we allocate resources in the constructor of a sub Timeline class, it’s best for us to release those resources in the destructor, and that the resources allocated in onBegin are best to be released in onEnd. For the resources allocated by onStart, things are a bit complicated; we can release the resources in onStop and onAbort. But if the resources allocated are used only by the internal business of the Timeline and not by any of its sub Timelines, we can release them in onInternalAbort and onInternalStop. This is a thumb of rule in resources management within Timeline life cycle; we’re free to do it on our own way as long as the resources can be properly allocated and released. Anyway, we need to understand the Timeline life cycle.

# Sub classing Timeline

Timelines prove to be of easy reuse and we have created Timeline subclasses dealing with common tasks in game programming. However, to fully exploit the power of Timeline mechanism, we need to extend the existing Timeline classes especially if we utilize Timeline to model game logic. That means we need to sub class Timeline and customize its behaviors.

## 4.1 Creating simple Timelines

Based on the knowledge we have acquired, we’re now ready to learn how to create a Timeline subclass. We start with a Timeline that only has its internal business; although we can add sub Timelines to it after it’s created, the sub Timelines is not part of the class thus has already been taken care of by Timeline class itself. So in Fig.2, the components dedicated to the managements of sub Timelines can be ignored for now. Examining the life cycle, we can find out that we can customize the behavior of Timeline internal business by overriding onStart, onInternalUpdate, onInternalAbort and onInternalStop.

Now let’s take some time to investigate the roles of these functions in Timeline life cycle. The main function that implements the Timeline life cycle is the update function in Timeline base class. It’s not a virtual function, thus Timeline life cycle is rather fixed. What we can do to customize the behaviors of a Timeline is to override the virtual functions with names on\*, which identify different phases in the Timeline cycle. By default, these functions do nothing, so if you create an instance of a Timeline base class; it has no internal business and can only serve as a container for other Timelines. The following paragraphs explain onStart, onInternalUpdate, onInternalAbort and onInternalStop in order.

The default implementation of onStart function in the base class does nothing. Since it’s called on the start of every single loop of Timeline, we can put initialization code like allocating resources, and initializing variables, etc. here for later use. If what the Timeline does is rather simple, you may not override it at all.

As you can see from the Timeline life cycle diagram, the onInternalUpdate function is the working horse of internal business. It’s called every frame when a Timeline starts, and returns a Boolean value to indicate whether its internal business has been finished. The return value of onInternalUpdate means whether the task should be continued, so if you have done the internal business, return false to signify that and return false otherwise. So we put the codes that update the internal business here to be called every frame, and determine whether the internal business has been done. By default, onInternalUpdate return false indicating no longer should it be called. So if the task you mean to do is simple, you can leave this function not overridden but instead finish all the tasks in the onStart function.

The onInternalAbort function is called when the internal business of a Timeline is aborted, which means that the Timeline is forced ending before onInternalUpdate returns false. You can regard it as an exception mechanism in Timeline life cycle. The codes that deal with the abort case may be different from those dealing with the normal stop case.

The onInternalStop function is called when the internal business of a Timeline is finished, which means that onInternalUpdate has returned false. When this function is called, it means that the internal business is finished as expected, otherwise onInternalAbort will be called. So we have to write codes that release resources allocated for internal business both in onInternalAbort and onInternalStop if we want the resources to be released when the internal business is aborted or finished. We demonstrate this with the following sample codes.

class FlipAtl : public Timeline

{

public:

FlipAtl(IWindow\* pFront, IWindow\* pBack, float fromRadian, float toRadian, float duration, const tstring& name = \_("FlipAtl"));

virtual ~FlipAtl();

void setAbortRadian(float fRadian, bool bUseAbortRadian);

protected:

virtual void onStart();

virtual bool onInternalUpdate(float dt);

virtual void onInternalAbort();

virtual void onInternalStop();

private:

void setRadian(float fRadian);

private:

float m\_fRunTime;

float m\_fDuration;

float m\_bUseAbortRadian;

float m\_fAbortRadian;

IWindow\* m\_pFront;

IWindow\* m\_pBack;

float m\_fFromRadian;

float m\_fToRadian;

};

FlipAtl::FlipAtl(IWindow\* pFront, IWindow\* pBack, float fromRadian, float toRadian, float duration, const tstring& name /\*= \_("FlipAtl")\*/)

: Timeline(name)

, m\_pFront(pFront)

, m\_pBack(pBack)

, m\_bUseAbortRadian(false)

, m\_fAbortRadian(0.0f)

, m\_fRunTime(0.0f)

, m\_fDuration(duration)

, m\_fFromRadian(fromRadian)

, m\_fToRadian(toRadian)

{

}

FlipAtl::~FlipAtl()

{

}

void FlipAtl::setAbortRadian(float fRadian, bool bUseAbortRadian)

{

m\_fAbortRadian = fRadian;

m\_bUseAbortRadian = bUseAbortRadian;

}

void FlipAtl::onStart()

{

m\_fRunTime = 0.0f;

}

bool FlipAtl::onInternalUpdate(float dt)

{

if (m\_fRunTime <= m\_fDuration)

{

m\_fRunTime += dt;

float ratio = m\_fRunTime / m\_fDuration;

float currentRadian = (1.0f - ratio) \* m\_fFromRadian + ratio \* m\_fToRadian;

setRadian(currentRadian);

}

return m\_fRunTime <= m\_fDuration;

}

void FlipAtl::onInternalAbort()

{

if (m\_bUseAbortRadian)

{

setRadian(m\_fAbortRadian);

}

}

void FlipAtl::onInternalStop()

{

setRadian(m\_fToRadian);

}

void FlipAtl::setRadian(float fRadian)

{

float cos\_x = cosf(fRadian);

m\_pFront->setVisible(cos\_x >= 0.0f);

m\_pBack->setVisible(!m\_pFront->isVisible());

m\_pFront->setScale(fabs(cos\_x), 1.0f);

m\_pBack->setScale(fabs(cos\_x), 1.0f);

**}**

Example 1

This code snippet is extracted from SD1 project with minor modifications. Its purpose is to simulate the flip animation using 2D scaling. We can see that the design of the class allow setting a different angle when the Timeline is aborted. It’s a simple class; by mimicking it you can start writing your first simple Timeline.

## Creating Timelines with Sub Timelines

The strength of Timeline comes from the way they can be composed in. We can program a very complicated progress from a higher level by creating a Timeline that makes use of other Timelines which model the sub progresses of the major one. This allows us to focus on the big picture other than mix codes that deal with details with those handling the main mechanism.

The simplest case is that the sub Timelines are independent to each other; so we can simply create a Timeline as container and add the sub Timelines according to their chronologicalrelations. This is the most common scenario in programming pure animations that do not interact with game logic. We do not have to create a new class for this. However, if we sub class Timeline to add codes that create, attach or remove sub Timelines dynamically, an instance of such Timeline can cooperate with its sub objects and acquires the ability to simulate a more dynamic progress. The real power comes from that other than write the internal business from scratch, a Timeline can delegate part of business to its subordinates and make the management of sub Timelines as part of its internal business. To achieve this, the Timeline may need to share resources (any object in the game could be considered as a resource) with sub Timelines. So the Timeline should allocate resources not only for its internal business but also its subordinates at specific phases in its life cycle and to release properly resources no longer used by it or its subordinates. To illustrate how to do that, we examine the roles of certain functions are playing from this aspect. These functions include the constructor, onBegin, onStart, onAbort, onStop, onEnd and destructor, which are to be studied respectively in the following paragraphs.

If the resources to be used are available when the Timeline is created, we can allocate it in the constructor. This approach has the advantage that the progress can be smoother when the Timeline starts running, because the blocking operations like file IO has been done when it’s created and may happen earlier in a loading progress. It may remind you of the RAII idiom and things are simpler if the initialization can be done in a contractor because it’s easier to maintain the integrity of the inner variables.

If the Timeline you write supports looping and the resources it uses do not change among the loops or the resources are not available when the Timeline is created, we can allocate it once in the onBegin function, and then reuse it until it’s no longer needed. If a Timeline is used to implement complex game logic, it’s rare, inconvenient and unnecessary to support loop, thus we can just forget onBegin and loops, but focus on using onStart. It’s also a good place for resources allocation that varies from loop to loop.

When onAbort is called, it means that either the internal business or the business of sub Timelines is aborted, thus some resources are no longer needed. Generally, it’s a best practice to release the resources allocated by onStart here. Similarly, we normally release these resources in onStop as well. We generally release the resources allocated by onBegin. And the best place to release the resources allocated by constructor is destructor.

We provide a sample code to illustrate how to create such Timeline.

class EPostWinAtl : public IPostWinAtl

{

public:

static EPostWinAtl\* createAtl(const Info& info, const tstring& name = \_SO("EPostWinAtl"));

public:

EPostWinAtl(const Info& info, const tstring& name);

virtual ~EPostWinAtl();

protected:

virtual bool onInternalUpdate(float dt);

private:

void trackCurWinLine();

void onCurWinLineSkipOrEnd(TimelineEvent& event);

private:

Timeline\* m\_pCurWinLineAtl;

int m\_nCurWinLineIndex;

};

EPostWinAtl\* EPostWinAtl::createAtl(const Info& info, const tstring& name /\*= \_SO("EPostWinAtl")\*/)

{

if (info.m\_SpinResult.m\_Payout.m\_nTotalWin == 0)

{

return nullptr;

}

return new EPostWinAtl(info, name);

}

EPostWinAtl::EPostWinAtl(const Info& info, const tstring& name)

: IPostWinAtl(info, name)

, m\_nCurWinLineIndex(-1)

, m\_pCurWinLineAtl(nullptr)

{

}

EPostWinAtl::~EPostWinAtl()

{

}

bool EPostWinAtl::onInternalUpdate(float dt)

{

int nWinLineSize = m\_Info.m\_SpinResult.m\_Payout.m\_vWinLine.size();

if (!m\_pCurWinLineAtl && nWinLineSize > 0)

{

m\_nCurWinLineIndex = (m\_nCurWinLineIndex + 1) % nWinLineSize;

ShowWinLineAtl::Info info;

info.m\_pSlotsMachine = m\_Info.m\_pSlotsMachine;

info.m\_WinLineData = m\_Info.m\_SpinResult.m\_Payout.m\_vWinLine[m\_nCurWinLineIndex];

tstring name = StringBuilder::format(\_SO("EPostWinAtL:ShowWinLineAtl\_#0")).add(m\_nCurWinLineIndex).build();

m\_pCurWinLineAtl = new ShowWinLineAtl(info, name);

trackCurWinLine();

addSubTimeline(m\_pCurWinLineAtl);

}

return true;

}

void EPostWinAtl::trackCurWinLine()

{

if (m\_pCurWinLineAtl)

{

m\_pCurWinLineAtl->registerEvent(TimelineEvent::TIMELINE\_SKIP\_BEGIN, this, CAST\_FUNC\_IEVENT(&EPostWinAtl::onCurWinLineSkipOrEnd));

m\_pCurWinLineAtl->registerEvent(TimelineEvent::TIMELINE\_END, this, CAST\_FUNC\_IEVENT(&EPostWinAtl::onCurWinLineSkipOrEnd));

}

}

void EPostWinAtl::onCurWinLineSkipOrEnd(TimelineEvent& event)

{

m\_pCurWinLineAtl = nullptr;

}

Example 2

This class is extracted from the Slot Editor project. Its purpose is to iterate and show each win line with animations after we spin and win in the Slots Machine emulator. It use a member variable m\_pCurWinLineAtl to identify a sub Timeline that plays animations for current win line. Two events are registered on the sub Timeline to track its state. The two events are TIMELINE\_END and TIMELINE\_SKIP\_BEGIN; the first one is triggered when a Timeline ends nor mather it’s aborted or stops normally; the second one is triggered when a Timeline is added to a parent Timeline but removed before it gets the chance to begin. When these 2 events are invoked, m\_pCurWinLineAtl is set to null to signify that the display task has been done so that in onInternalUpdate, the parent Timeline knows it’s time to iterate the next win line and create the sub Timeline playing the animations and then add it. m\_pCurWinLineAtl is used like a weak reference here. Tip:pleae pay attention that the approach of registering TIMELINE\_END and TIMELINE\_SKIP\_BEGIN is very common in the codes that make use of Timeline. Listening to both events make sure the state of a Timeline is properly tracked.

You may have noticed that we could use the loop function of Timeline to implement the above logic. SD1 adopts this approach for the same feature, which is more complicated, but it’s not suggested to encapslate complicated logic in a looped Timeline. To write codes that support loops of complicated game logic, you have to take into account many charascterics of Timeline machesism and features of logic itself and be very cautious about resources management. Please do not put a complicated Timeline inside a loop unless you’re very farmilar with the Timeline implemtation.A good approach is that only use loop for Timelines of pure animations.

# Name and Id

Timeline has a name property and

# What is Timeline not good at

# The future of Timeline