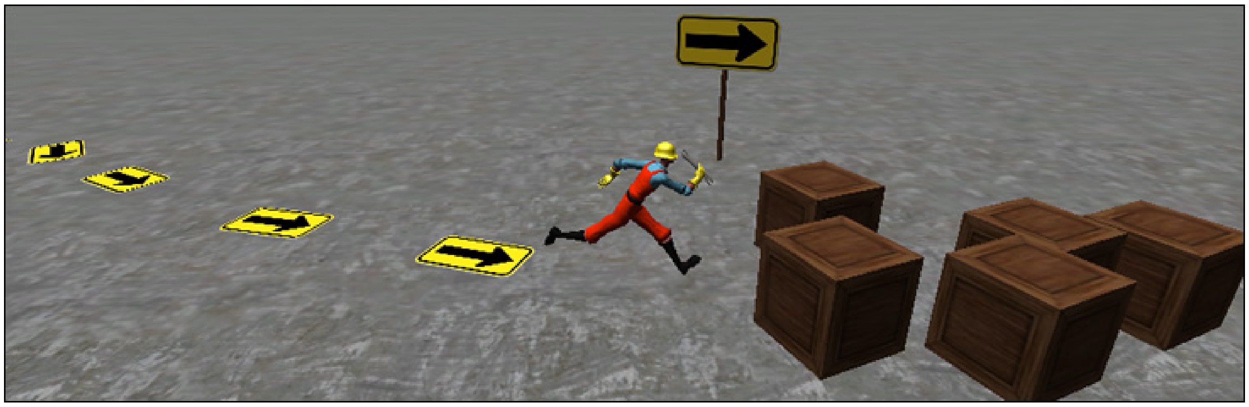
Game replay data

movies, rewinding and data mining



Made for Creative Technology, Universiteit Twente & Multimedia @ VU Amsterdam
by Mike Hergaarden
October 2010

Table of contents

Introduction

This document describes my implementation of game replays. I distinguish three types of game “replays”:

1. Replay without interaction: See a replay movie of a series of events in a game. I’ll refer to this as “**Replay movies**”.
2. Replay with interaction: Rewind the game and redo your actions to change the result. Called “**Rewind**” from now on.
3. Being able to “replay” user data: gather user interaction data to be able to investigate your users actions and behaviour. This is called “**Data mining**”.

In this document I describe my example project in which all three of the techniques are possible. I discuss the Pros and Cons of my implementations and offer you insights to add any of the three replay methods to your own applications. This document comes with the full sourcecode of my implementation. It is advised to explore the Unity project while reading this document. It is available on m2h.nl/unity.

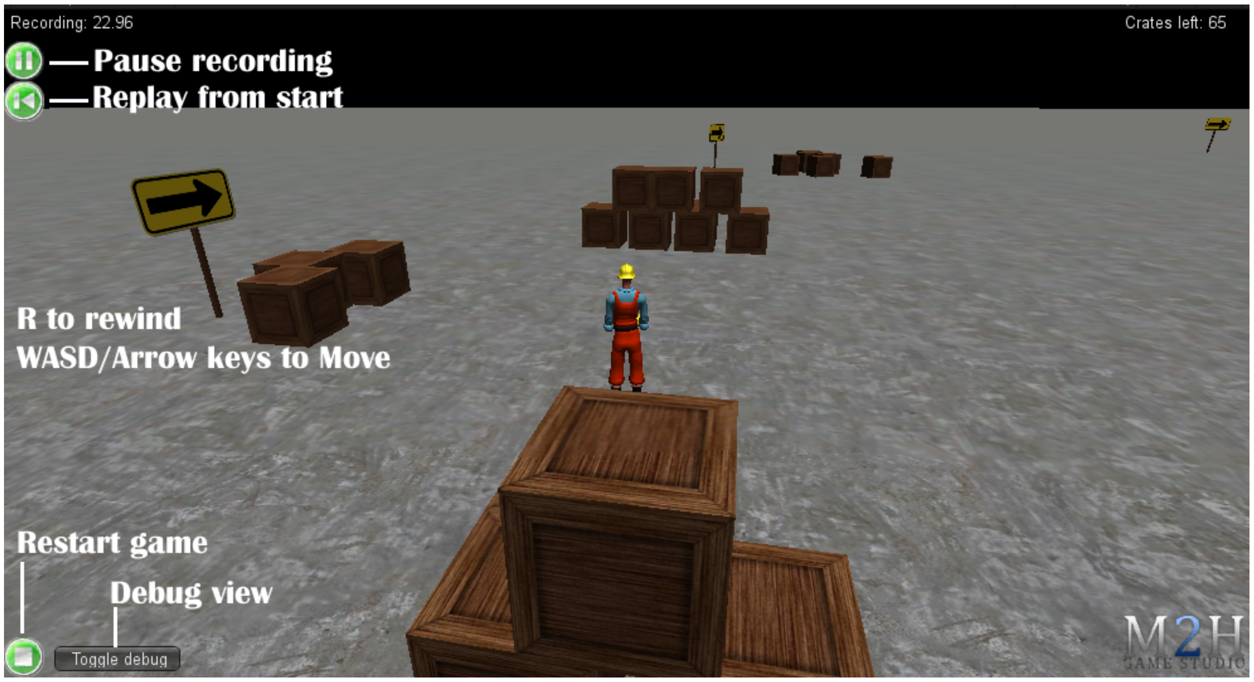
An example of a replay movie:
Halo 2 is one of the many games that allow users to view a repay movie of the game session you’ve just finished. This movie can be shared for bragging rights or analyzed to improve your tactics.

An example of rewinding:
Briad (http://braid-game.com) is a game without lives or game-overs. When your game character dies you rewind the game a few seconds to change the course of events.

An example of data mining:
Data mining has been around very long. However, data mining did not make a big appearance in games until the success of Farmvillle. Farmville tracks all user interactions in order to maximize the odds of a player making a micro transaction.

Unity implementation

First, I ask you to open and run the replay scene. Try walking around for a bit and 'destroy' some crates by walking over them, use the right mouse button to place a 'poster' on the floor .Now the fun starts: experience the game replay movie and rewinding features. Press **R** to rewind your game or use the GUI buttons to view replays of your game. After having played around with the project for a bit we'll go over the implementation details.



The replay project

The easiest approach to realize a replay feature is by saving every objects state for every frame in the game. For a player you'd save it's position, rotation, attributes(health), etc. every frame. However, this would sum up to Gigabytes of data per game session. To reduce the amount of saved data you should save only changes in data. I refer to these change points as 'keyframes'. Note that the data changes you need to save are quite similar to the data set you need to synchronize in a multiplayer game. This could aid your multiplayer game design or vice versa.

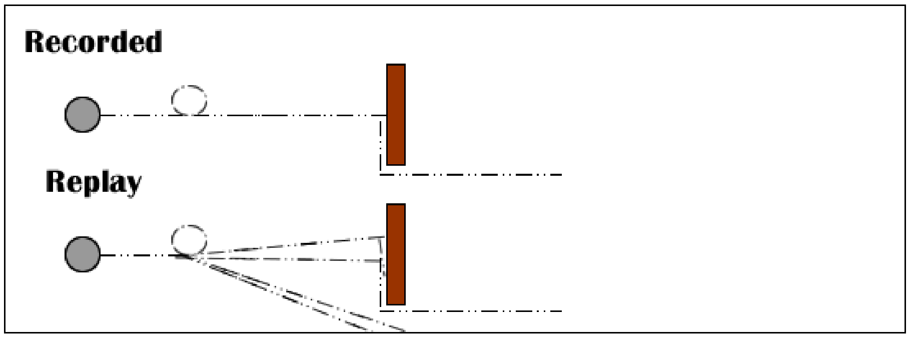
How I got to the end result

The best way to explain the fundamentals of the replay project is to explain you how I got to the final project. This way you'll fully understand why and how I made design decisions. Even if you disagree with these decisions.

Attempt 1: Save player state changes only - Doesn't work

In my first attempt I wrote down the events that happened that matter for replay data. In the example project you control a simple character with basic third person movement. So a few of events that matter are "*StartMovementSideRight, StopMovementSideLeft, StartMovementSideLeft, ..*". I quickly realized this should be simplified to more basic states.

Attempt 2: Save player control changes only - Incorrect replays



Attempt 2: save control changes only

For player controls it all comes down to the following changes: Which movement key is pressed (or released) and at which time. E.g.:

```
6.300717: forwardMovement (start/stop forward or back)
6.556254: forwardMovement (start/stop forward or back)
7.596561: sideMovement (start/stop left or right)
8.574206: jump (start/stop)
8.587856: jump (start/stop)
```

Both the beginning and end of an event is saved. Note that the event was saved as int instead of a string.

The difference with attempt 1 is that there's not a start/stop forward and a start/stop backwards, but just one 'forwardMovement' event saved with its value (1, 0, ,-1). This worked pretty well at first and it worked very smooth when replaying the data. However, the replay was incorrect and lost accuracy with every event. This happened because even small time variations matter for the physic calculations. A rotation could be off by a few milliseconds and thereby drastically changing the following steps.

Attempt 3: Bugfix attempt 2

Since attempt 2 wasn't that bad, I thought that by tweaking this with a few 'hacks' I could solve all problems. To compensate for the replay using invalid values, every event/'keyframe' should include the correct rotation, position and movement direction at that time.

The data now changed to the following format:

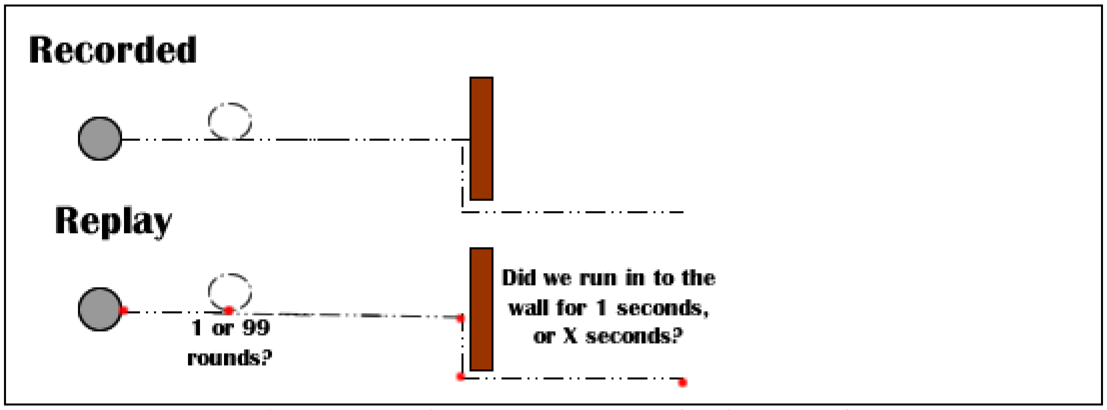
```
6.300717#0#1#0.71_1_4.72#0_108.04_0#0.95_0_-0.31
6.556254#0#0#-0.26_1_5.03#0_288.04_0#-0.95_0_0.31
6.764966#1#-1#-0.82_1_5.21#0_288.04_0#-0.95_0_0.31
7.596561#1#0#-0.95_1_5.06#0_15.65_0#0.27_0_0.96
8.574206#2#1#-0.72_1_5.89#0_15.65_0#0.27_0_0.96
```

```
8.587856#2#0#-0.72_1_5.89#0_15.65_0#0.27_0_0.96
```

Time # EventType # EventValue # Position # Rotation # Direction

The data is now always correctly displayed, but in between the events some lag would occur. Overwriting the replay with the correct values is clearly visible. Adding some smoothing (lerping) could help, but would be just another hack. The root of the inaccuracy problem should be solved differently instead.

Attempt 4: Save position/rotation values at movement changes only and calculate the required movement at runtime



Final setup: By design, you need to take care of one issue yourself.

It is important to note that the required replay data will always depend on the ontology/domain of your application. My goal was to create a platform that could be customized for any game. Therefore I made an important design decision: The replay data should be used to recalculate the correct movement when running the replay. This allows for a very smooth replay, furthermore: data storage is very 'cheap'. There is one caveat here: There's a bit more work in recalculating the required movement, and if not done right, the movement could look a bit different.

Two examples where the recalculation of the required movement is not so obvious are displayed in the image of Attempt 4. Using just regular position and rotation 'keyframes', we are unable to predict whether something moved 360 degrees or 720 degrees. Furthermore it is impossible to predict whether a player turned left or right. There also other non-rotation related cases such as running against a wall for 30 minutes: there would be no movement in the replay. Fixing these issues simply requires you to set more regular keyframes, or save additional data. I.e. to fix the rotation issues, simply add a keyframe after every 179 degrees rotation.

To conclude

Deciding which data I should use for the replays was a bit of a puzzle. I started with several attempts that recorded the actual player movement. To improve the precision of replays I also saved the position and rotation. However this did not always yield perfect results and since position and rotation could be all that's required to replay, I decided that recalculating the required movement at runtime (during a replay/

rewind) would be the best solution for this example. This saved a lot of data and the replay works without any hitches.

How the replay function works

The actions that matter in this example game are:

1. Player movement (position, rotation)
2. Crate pickups
3. Placing posters

Crate pickups are handled by the collision with the player via the CrateCollision script. On collision it spawns particles and disables the crate just as you would in a normal game. For replay purposes this line is added:

```
    CrateCollision.cs line 16
RecordGameplay.SP.AddAction(RecordActions.crateCollision, collider.transform.position);
```

We maintain a list of all actions in the game to be able to replay them. The RecordGameplay class maintains it's own timer and timestamping. For player movement we do the same thing: Every time the horizontal *or* vertical movement changed we add a "keyframe":

```
    ThirdPersonControler.cs line 418 & 404
RecordGameplay.SP.AddAction(RecordActions.playerMovement, transform.position, transform.localEulerAngles);
```

Note that we didn't add any rotation value to the crates action as there only the position matters. Finally the poster creation is recorded via:

```
  ThirdPersonControler.cs line 228
RecordGameplay.SP.AddAction(RecordActions.createPoster, pos, rot);
```

The player position is saved about, say, 2 times a second (depends on the actual movement). Crate actions are triggered just once when picked up and the poster events are triggered very time you place a poster.

The RecordGameplay class is designed to be reusable without (much) changes. That's why I placed all gameplay specific implementation code into Replay.cs. The RecordGameplay class is the core of this project, it's full source code has been included in Appendix A. For this example game, all calls to RecordGameplay have been integrated in the Replay.cs class (besides the calls to `RecordGameplay.SP.AddAction();`). All there is to the actual replay code is shown below, this is an excerpt of the actual replay related code from Replay.cs. Rewinding and GUI have been left out.

```
public enum RecordActions { playerMovement, crateCollision, createPoster }

  //  Actual replay code
   //
   private float replayStarted = 0;

   void StartReplay()
   {
       viewingReplay = true;

       //Reset player character
       Destroy( GameObject.FindGameObjectWithTag("Player") );
```

```
        SpawnCharacter();
        ThirdPersonController.SP.isReplaying = true;

        //Reset objects
        CrateCollision.EnableAllCrates();
        DestroyAllPosters();

        //Start replay
        replayStarted = Time.realtimeSinceStartup;
        StartCoroutine(RunReplay());
    }

    void StartReplayFrom(float timepoint)
    {
        Debug.Log("StartReplayFrom" + timepoint);
        recorder.PauseRecording();
        StartReplay();
        replayStarted -= timepoint; //Skip some time.: All skipped actions will be applied right away
    }

    public static bool IsReplaying(){
        return SP.viewingReplay || SP.rewinding;
    }

    float GetReplayTime()
    {
        return Time.realtimeSinceStartup - replayStarted;
    }

    IEnumerator RunReplay() //Apply  all events that have passed
    {
        int i = 0;
        List<RecordedEvent> actions = RecordGameplay.SP.GetEventsList();
        foreach(RecordedEvent action in actions){
            while (viewingReplay && action.time > GetReplayTime())
                yield return 0;
            if (!viewingReplay) break;

            //Apply action
            RecordedEvent nextAction = GetNextAction(action, actions);
            ApplyAction(action, nextAction, false);
            i++;
        }
        StopReplay();
    }
  // Get the next RecordedEvent that has the same "RecordActions " as curAction. E.g. the current & next movement events.
    RecordedEvent GetNextAction(RecordedEvent curAction, List<RecordedEvent> actions)
    {
        RecordActions actionType = curAction.recordedAction;
        bool foundCurrent = false;
        foreach (RecordedEvent action in actions)
        {
            if (foundCurrent)
            {
                if (actionType == action.recordedAction)
                    return action;
            }
            if (action == curAction)
                foundCurrent = true;
        }
        return null;
    }

    void StopReplay()
    {
        ThirdPersonController.SP.isReplaying = false;
        viewingReplay = false;
        recorder.StartRecording();//Continue recording right away
    }
```

```
//This has to be customized per game: Apply your specific actions.
   void ApplyAction(RecordedEvent action, RecordedEvent nextAction, bool rewind)
   {
      //Debug.Log("Action="+action.action+" pos="+ action.position + " next=" + nextAction);
      if (action.recordedAction == RecordActions.playerMovement)
      {
         if (nextAction == null)
            ThirdPersonController.SP.SetReplayData(Time.time, action.position, action.rotation, Time.time, action.position, action.rotation);
         else
         {
            float nextTime = nextAction.time - action.time;
            if(rewind)
               nextTime = action.time - nextAction.time;
            ThirdPersonController.SP.SetReplayData(Time.time, action.position, action.rotation, Time.time + nextTime, nextAction.position,
nextAction.rotation);
         }
      }else if (action.recordedAction == RecordActions.crateCollision){
         if (rewind)
            CrateCollision.EnableCrate(action.position);
         else
            CrateCollision.DisableCrate(action.position);
      }else if (action.recordedAction == RecordActions.createPoster){
         if (rewind)
            RemovePoster(action.position, action.rotation);
         else
            SpawnPoster(action.position, action.rotation);
      }
   }
```

How the rewinding works

With the replay in place, rewinding was really easy to add. It required just two tweaks as I could reuse the replay functionality as base for the rewinding; I only had to invert the time. Futhermore; when stopping a rewind all actions that happened after the end time point had to be undone in the game. They are automatically removed from the replay list in the RecordGameplay class.

```
   // REWIND CODE
   //
   public bool rewinding = false;
   public float startRewind;

   float GetRewindTime()
   {
      return Mathf.Clamp(  (recorder.LastRecordLength() -  (Time.realtimeSinceStartup - startRewind)), 0, recorder.LastRecordLength()  );
   }
   void StartRewind()
   {
      if (rewinding) return;
      rewinding = true;

      recorder.PauseRecording();
      startRewind = Time.realtimeSinceStartup;

      Debug.Log("StartRewind: Time=" + Time.realtimeSinceStartup + "  startRewind=" + startRewind + "    from " + GetRewindTime() + "
recorder.LastRecordLength() =" + recorder.LastRecordLength());

      ThirdPersonController.SP.isReplaying = true;
      StartCoroutine(RunRewind());
   }
   IEnumerator **RunRewind**()
   {
      List<RecordedEvent> actions = RecordGameplay.SP.GetEventsList();
      for (int i = actions.Count-1; i >= 0; i--)
      {
         RecordedEvent action = actions[i];
         while (rewinding && action.time < GetRewindTime())
            yield return 0;
         if (!rewinding) break;
         //Apply action
         RecordedEvent nextAction = GetNextRewindAction(action, actions);
         ApplyAction(action, nextAction, true); //Re-used from Replay
      }
      StopRewind();
   }

//Get the next "RecordedEvent" entry with the same RecordActions enum value (e.g. the next playermovement AFTER curAction)
   RecordedEvent **GetNextRewindAction**(RecordedEvent curAction, List<RecordedEvent> actions)
   {
      RecordActions actionType = curAction.recordedAction;
      bool foundCurrent = false;
      for (int i = actions.Count - 1; i >= 0; i--)
      {
         RecordedEvent action = actions[i];
         if (foundCurrent)
         {
            if (actionType == action.recordedAction)
               return action;
         }
         if (action == curAction)
            foundCurrent = true;
      }
      return null;
   }

   void **StopRewind**()
   {
      if (!rewinding) return;
      rewinding = ThirdPersonController.SP.isReplaying = false;
      //The recorder will delete all actions after this end time, therefore:
      //Undo some gameplay changes
      List<RecordedEvent> replayData = recorder.GetEventsList();
      if (replayData.Count >= 0)
      {
         for (int i = replayData.Count - 1; i >= 0; i--)
         {
            RecordedEvent action = replayData[i];
            if (action.time >= GetRewindTime())
            {
               if (action.recordedAction == RecordActions.crateCollision)
                  CrateCollision.EnableCrate(action.position);
            }
         }
      }
      recorder.StartRecording(GetRewindTime()); //Continue recording again :)
   }
```

Customizing the replay project

The replay project features two main classes: Replay.cs and RecordGameplay.cs. Besides these two files there is the player control script, a camera script and a crate pickup script. RecordGameplay is the most important class as it's designed to be a general base for your own customization, only little customization could be required. Replay is the script that needs to be fully customized to your own needs. Furthermore you should add calls to the RecordGameplay from your own objects (player etc.).

To use and customize the Replay script for your own projects
1. Copy the Replay.cs and RecordGameplay.cs scripts to your own project

2. Decide the types of data you need to track. Think about all possible changes (Crate picked up, health lost), etc. Now customize the RecordActions enum in Replay.cs to include all your games unique actions. The examples actions are:

```
public enum RecordActions { playerMovement, crateCollision, createPoster }
```

3. Notify RecordGameplay whenever a change occurs, this requires just one line per action/object. Hereby use the enum from step 2.

```
RecordGameplay.SP.
AddAction(RecordActions.playerMovement,  transform.position, transform.localEulerAngles);
```

4. Remove the examples gameplay specific lines from Replay.cs and replace/add your own gameplay. Decide if you want to reuse the GUI options?

As an example, the crate pickups are tracked via just one line:

```
void OnControllerColliderHit(ControllerColliderHit hit) {
    Collider collider = hit.collider;
    if (collider.tag == "Destroyable")
        RecordGameplay.SP.AddAction(RecordActions.crateCollision, collider.transform.position);
}
```

Here the crates are tracked via their unique positions.

Data mining

Why use datamining?

An interesting read is the article "Better Game Design Through Data Mining" by David Kennerly on Gamastura. This article discusses how data mining can be used in an MMO type game for game balancing, anti-cheating, cutting costs and keeping your players. Kennerly put it simply: "Why Mine Data? Because players lie.".

How to start mining?

While data mining is not present in the example project, all code to do so is there. The replay project is perfectly usable for data mining purposes. The easiest use case would be to upload a users recorded replay data to your own webserver like so:

```
public IEnumerator UploadData()
   {
      WWWForm wwwForm = new WWWForm();
      wwwForm.AddField("replayData", RecordedDataToString());
      WWW www = new WWW("http://www.YOURSITE.com/uploadData.php", wwwForm);
      yield return www;
      Debug.Log("Uploaded replay data!");
   }
```

A reminder of the default RecordedDataToString() output format:

```
Time # EventType # Position # Rotation
e.g.: 8.587856#2#-0.72_1_5.89#0_15.65_0
```

Your server could receive the data like so:
PHP example - receive the data on a webserver

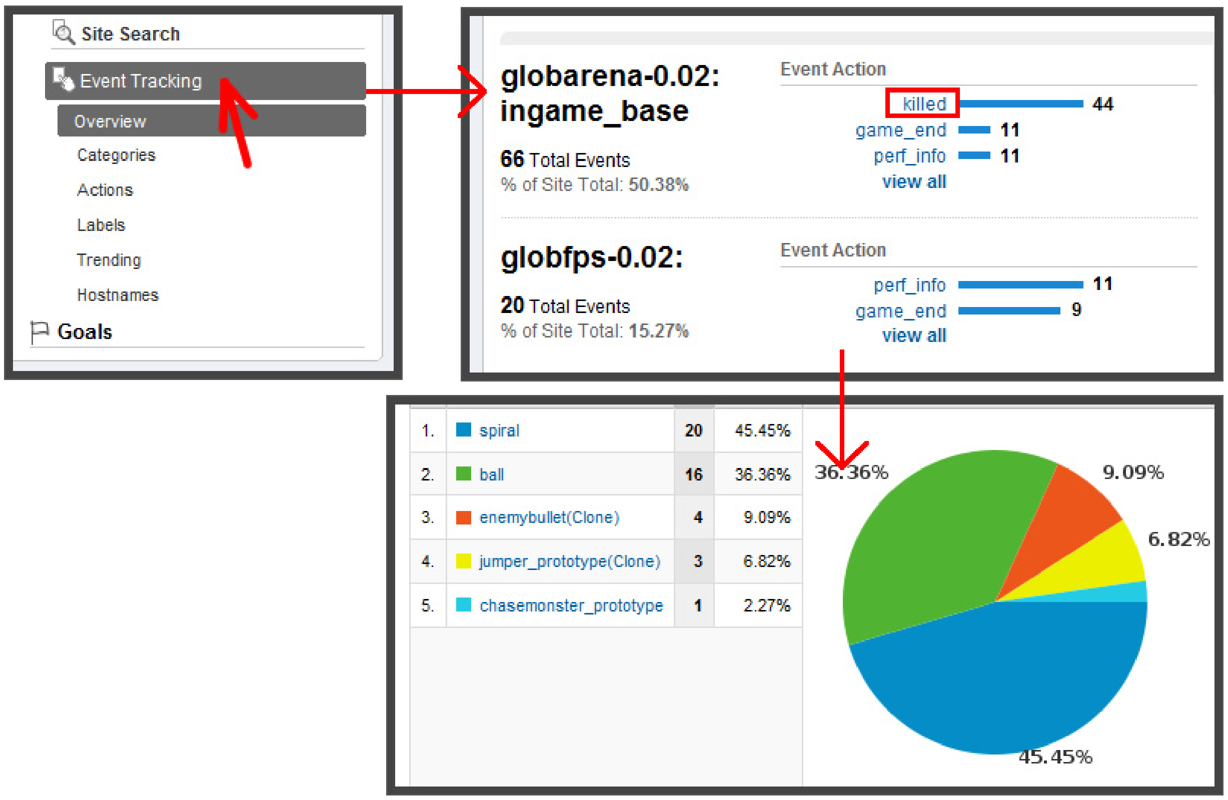
```
<?php
$replayData = $_REQUEST['replayData'];
$lines = explode("\n", $replayData);

foreach ($lines as $singleLine) {
              $values = explode("#", $singleLine);
              $time = $values[0];
              $eventType= $values[1];
              $position = $values[2];
              $rotation = $values[3];
             //Use the data here...
}
?>
```

You can parse this data however you like. Don't forget to filter out only the data you (could possibly) care about. If you customize the replay data to contain strings and such you’d better upload the data in JSON format to prevent bugs. I advice you to use the LitJSON plugin in that case (http://litjson.sourceforge.net).

Google analytics integration in Unity

“Google Analytics lets you measure your advertising ROI as well as track your Flash, video, and social networking sites and applications.” -http://www.google.com/analytics/

For an even easier, but limited, data mining implementation you could make use of Google analytics. Using Google analytics has the advantage that it does not require you to run and create your own data and data reporting back-end. The only downside is that you depends on the Analytics API and thus less customization is possible.



globarena-0.02: ingame_base

66 Total Events
% of Site Total: **50.38%**

Event Action	
killed	44
game_end	11
perf_info	11
view all	

globfps-0.02:

20 Total Events
% of Site Total: **15.27%**

Event Action	
perf_info	11
game_end	9
view all	

1.	🟦 spiral	20	45.45%
2.	🟩 ball	16	36.36%
3.	🟧 enemybullet(Clone)	4	9.09%
4.	🟨 jumper_prototype(Clone)	3	6.82%
5.	🟦 chasemonster_prototype	1	2.27%

A perfect example of datamining in a game is shown above. This image shows a Google analytics data example from mostlytigerproof.com. The pie chart shows the causes of death in all game sessions (“killed” event). This data can be used to improve the game by, for example, ensuring players don’t quit the game after dieing 10 times in a row.

Unity datamining implementation by Craig Timpany:
http://blog.mostlytigerproof.com/2009/10/06/gathering-statistics-using-google-analytics-and-unity-3d/

Conclusion

Replay and rewinding implementations depend on a games ontology and will always require some customization per game. My platform should provide a solid base for replay projects. When implementing replay functionality the toughest design decision is deciding what data needs to be stored and how the replay data is used. This is essential to reducing data storage and replay accuracy.

This document has shown you my Unity replay implementation. It has explained you how to customize it for your own purposes and taught you about replay, rewinding and data mining in general. I hope you are now ready to apply these resources for your own needs.

Appendix A: RecordGameplay.cs

As a reference, here is the full sourcecode of RecordGameplay.cs. This class is the core of recording your gameplay. It is independant from your gameplay logic and can be reused for your own purposes.

```
using UnityEngine;
using System.Collections;
using System.Collections.Generic;

public enum RecordStatus { stopped = 1, paused, recording }

public class RecordGameplay : MonoBehaviour {

    public static RecordGameplay SP;
    private RecordStatus recordStatus;
    private float startedRecording =0;
    private List<RecordedEvent> replayData = new List<RecordedEvent>();
    private float pausedAt = 0;

   void Awake(){
      SP =this;
      StartRecording();
   }

   public float RecordTime()
   {
      if (recordStatus != RecordStatus.recording)
          Debug.LogError("Cant get time!");
       return Time.realtimeSinceStartup - startedRecording;
   }

    public void PauseRecording()
    {
        if (recordStatus != RecordStatus.recording)
            Debug.LogError("Cant pause! " + recordStatus);
        pausedAt = RecordTime();
        recordStatus = RecordStatus.paused;
    }

    public void StartRecording(float startTime){
        //Delete all actions after STARTTIME. Continue Recording from this point
        if (replayData.Count >= 0)
        {
            for (int i = replayData.Count - 1; i >= 0; i--)
            {
                RecordedEvent action = replayData[i];
                if (action.time >= startTime)
                    replayData.Remove(action);
            }
        }
        pausedAt = startTime;
        StartRecording();
    }

    public void StartRecording()
    {
        if(recordStatus == RecordStatus.paused){
            startedRecording = Time.realtimeSinceStartup - pausedAt;
        }else{
```

```
        startedRecording = Time.realtimeSinceStartup;
        replayData = new List<RecordedEvent>();
    }

    recordStatus = RecordStatus.recording;
}

public bool IsRecording(){
    return recordStatus == RecordStatus.recording;
}
public bool IsPaused(){
    return recordStatus == RecordStatus.paused;
}

//Hereby multiple defenitions so that you can add as much data as you want.
public void AddAction(RecordActions action)
{
    AddAction(action, Vector3.zero);
}
public void AddAction(RecordActions action, Vector3 position)
{
    AddAction(action,  position, Vector3.zero);
}
public void AddAction(RecordActions action, Vector3 position, Vector3 rotation)
{
    if (!IsRecording())
    {
        //Debug.LogError("Record didn't start!");
        return;
    }

    RecordedEvent newAction = new RecordedEvent();
    newAction.recordedAction = action;
    newAction.position = position;
    newAction.rotation = rotation;
    newAction.time = RecordTime();
    replayData.Add(newAction);

            }

public void StopRecording()
{
    if(recordStatus != RecordStatus.recording)
        Debug.LogError("Cant STOP!");

    stoppedAtLength = RecordTime();
    recordStatus = RecordStatus.stopped;
}

private float stoppedAtLength = 0;
public float LastRecordLength()
{
    if (recordStatus == RecordStatus.paused) return pausedAt;
    if (recordStatus != RecordStatus.stopped) return RecordTime();
    return stoppedAtLength;
}

public string RecordedDataToReadableString(){
    string output ="Replay data:\n";
    foreach(RecordedEvent action in replayData){
        output+= action.time+": "+action.recordedAction+"\n";
    }
    return output;
}
public string RecordedDataToString()
{
    string output = "";
    foreach (RecordedEvent action in replayData)
    {
            output += action.time + "#" + (int)action.recordedAction + "#" + Utils.Vector3ToString(action.position) + "#" +
Utils.Vector3ToString(action.rotation) + "\n";
        }
        return output;
    }

    public List<RecordedEvent> GetEventsList(){
        return replayData;
    }


    /* //Add the right URL to your upload script
     public IEnumerator UploadData()
    {
        WWWForm wwwForm = new WWWForm();
        wwwForm.AddField("replayData", RecordedDataToString());
        WWW www = new WWW("http://www.YOURSITE.com/uploadData.php", wwwForm);
        yield return www;
        Debug.Log("Uploaded replay data!");
    }
     */
}

public class RecordedEvent {
    public RecordActions recordedAction;
    public float time;
    public Vector3 position;
    public Vector3 rotation;
}
```