Fresh Play Statistics

A need for every game is to gather play statistics during white box tests (at least). Play statistics allow the developer to gather crucial data from the field about how players are actually interacting with the game. This data can then be used to improve the game.

The kinds of data that a developer might want to gather for a particular game are diverse.

* The time and location of every mouse click or tap.
* The position of the mouse at all times.
* The location where the player’s eyes are fixed (assuming software and hardware is available to measure this).
* In-world locations where the player died or took damage.
* The length of time taken in a particular level or section of a level.
* The number of times the player clicked various buttons.
* The circumstances in the game when the player quit.

The types of data gathered for a particular game are diverse; the types of data that might be gathered across *all* games are vastly diverse. Therefore a system designed to gather, transmit, store, and analyze play statistics must be capable of dealing with data at an abstract level.

Yet in order to be useful, the system must also have concrete, powerful mechanisms for quickly gathering and analyzing data.

My current goal is to build a play statistics (telemetry) system for *House of Shadows* (summer 2012). But because I’ve build a few such systems before, and because I foresee needing them again in the future, and because building such a system takes enough time that their creation tends to get delayed, I’d like to build enough core functionality that I can more readily create telemetry systems for future games.

The design challenge for the system comes from the tension between three opposing goals:

1. Provide an abstract, highly configurable system.
2. Provide a concrete, highly powerful and immediately useful system.
3. Use as little bandwidth and server space as possible.

The overall architecture of my system is clear.

1. A per-game, per-test MySQL database is configured on a web server.
2. One or more PHP scripts are implemented to send data to the database (during the data gathering phase) and to view and analyze data (during the analysis phase).
3. A client-side API allows the game to conveniently send data to the server.
4. The game itself detects significant UI events and uses the API to send them.

Fresh is responsible for the first three items in a game-neutral way. The individual game is responsible for deciding when to upload data and what to send (item #4). Additionally, the individual game supplements item #2 with its own analysis techniques, using whatever tools (Excel, Mathematica, additional MySQL queries and/or PHP scripts) it chooses.

# The Essential Concepts

It makes sense to center the system around the idea of an “Event.” An event is a discrete occurrence in time, with a particular Type and potential additional data. Examples:

* Event PlayerDeath occurred at session time 00:24:49.23 in map “level02.xml” and location (421,681).
* Event ButtonDown occurred at session time 00:00:05.24 in context “MainMenu” for button “Play”.

Using this kind of data, the analyzer could then measure on average how long it took for users to tap the “Play” button. He could build a heatmap showing the locations in a given map where deaths occurred. Overall, he could read the “story” of a user’s interaction with the game.

The inalienable data that must be associated with any event are:

* Project
* Platform
* Version/Build
* User (by user name, IP address, or whatever)
* Play Session
* Session Time

# Frequency of Transmission

The major challenge from a performance and reliability standpoint is when to actually send data from the client to the server. There are two extreme options.

1. Whenever an event is registered on the client.
2. At the end of a session.

Intermediate options include sending data at the close of certain *contexts* and, more interestingly, “flushing” the event queue when it grows to a certain size.

The advantage to harboring data on the client (e.g. queuing events and occasionally flushing) appears only if the data can be compressed. For example, if a single “report” lists the session Id only once for the whole report rather than once per event, this saves data on the order of the size of the session Id times the number of queued events.

Considering the structure I propose below, however, I doubt any real savings from this kind of compression would be forthcoming. I’m simply not using enough strings nor sending enough redundant data to make a difference.

So I will send the data with every event.

(This will change if the actual act of sending the data is expensive on the client such that it causes a hitch. In that case, an explicit flush() command timed by the client to coincide with level loads, etc., would be preferable.)

# Structures

Although likely to be implemented as database tables, the following C-style structures imply the data and relationships associated with each type.

struct User

{

string name;

string lastIPAddress;

}

struct Session

{

id user;

DateTime enterTime, exitTime;

id resumingSession = NULL;

}

struct PlayContext

{

string name;

id session;

id parentContext = NULL;

DateTime enterTime, exitTime;

}

struct Event

{

id playContext;

DateTime sessionTime;

string type;

id subject;

string comment;

}

struct EventLocalized : Event

{

float x, y, z;

}

struct Subject

{

string type;

string name;

}

# Server API

Although implemented as one or more PHP pages taking POST or GET variables, the following C-style function declarations describe the interface for the database.

id startSession( string username );

void endSession( id session );

void pauseSession( id session ); // Records e.g. iPhone task switching

void resumeSession( id session );

id enterContext( string name );

void leaveContext( id context );

void registerEvent( string type, string subjectType, string subjectName, string comment );

void registerEventLocalized( string type, string subjectType, string subjectName, float x, float y, float z, string comment );