Smoothernity

June 4, 2014

Contents

1 Idea **2**

2 Generation **2**
- 2.1 Primitives Level . . . 2
- 2.2 Concepts Level . . . 2
- 2.3 World Level . . . 3

3 Runtime **3**

4 Editing **4**

5 Scale **4**

6 Grid **5**

7 Multiplayer **5**

8 Implementation **6**
- 8.1 Platform-Dependent Part . . . 6
- 8.2 Platform-Independent Part . . . 6
- 8.3 Scripts Sandboxing . . . 6

9 Assorted Notes **7**

1 Idea

The idea is to create a program which sports the following features:

1. Huge immersive virtual universe[1].
2. Low physical storage space consumption[2].
3. Real-time universe generation[3].
4. Constant frame rate[4].
5. Ports for different platforms[5].
6. Effective utilization of the available resources on each platform[6].
7. Interactive universe editor[7].
8. Small platform-dependent code part[8].

2 Generation

Whole generation process can be split into several levels of abstraction.

2.1 Primitives Level

Generates reusable low-level building blocks. Implemented in code, optimized for each platform. Heaviest computations should be performed on this level. The examples of primitives are: "marching cubes of function $f(x, y, z)$", "perlin noise landscape", "spline surface", etc. It operates with such entities as "triangles", "meshes", "lights", "shaders", "OpenCL kernels", "collision objects", "rigid bodies", "sound waves data", etc.

2.2 Concepts Level

Generates reusable high-level building blocks from other reusable building blocks (both high- and low-level). Implemented in platform-independent code. It composes primitives and other concepts into higher-level structures, such as "character", "vehicle", "building", "city", etc.

[1] E.g. from the scale of stars down to planets, buildings and rooms.

[2] E.g. space on hard drive(s) of local or remote machine(s). Essentially, this means low *Kolmogorov's comlpexity* of the program.

[3] The world seamlessly generates on-the-fly as player moves through the universe.

[4] Smooth visual animation without interruptions during the whole time the program is running.

[5] Both desktop and mobile.

[6] That is, it should yield "better picture" on "faster platforms".

[7] The designer can navigate through the universe, make changes and see the result as soon as possible.

[8] Let's say, $\leq 10,000$ lines of code for each port. E.g. it's OK to have many ports with large *cumulative* size as long as none of them exceeds this limit when counted separately.

2.3 World Level

Generates unique content from high-level building blocks. Implemented as data. Connects concepts into directed acyclic graph of dependencies and specifies attribute values[9]. This graph can be edited either interactively by the designer or programmatically through "game script"[10].

Concepts implementation code is itself a dependency for every instance of this concept in the dependency graph. Thus, when concept code is changed, all the affected instances and their dependents can be regenerated. Implementation-wise, one possible way to do this may be by tracking all primitives created by the particular concept, so that when this concept is changed, all primitives can be disposed automatically.

This reloading idea in general case is applicable for static objects only. That is, object state must depend only on other objects' states, but not on the self state in a previous moment in time. It won't work for dynamic case, e.g. because relative phase between objects' states is changed due to reload.

For dynamic case, we can edit the initial state with frozen time[11] at 0-time. Then start simulation by clicking "play" button. To continue editing, we "pause" again, reset to 0-time.

Game scenario state can be represented by a vector with integer components. This vector is essentially an input for world generation. Some nodes behave conditionally to the game scenario state. To check game at various stages it's sufficient to change this vector. Affected nodes can be regenerated on-the-fly, just as regular dependencies.

3 Runtime

Runtime state is the most volatile part of game state. It's not persistent, it exists only while game runs. Generation part (Section 2) defines a shape of the universe. Runtime part defines its behavior.

Runtime part depends on generation part[12]. Every resource generated by generation part, is owned by generation part. Whenever generation part changes[13], runtime part restarts. That is, all game entities spawned in runtime part[14] are discarded, memory is cleared and runtime script is restarted from scratch. After restart, runtime part can quickly retrieve resources that were already generated before restart[15].

Runtime part can change game scenario vector, thereby effectively restarting itself. This is how game script is implemented. It should be possible to do this restart seamlessly, within

[9] E.g. city of 20 buildings with $3 \ldots 5$ rooms each.

[10] E.g. add city C to planet P after the player had picked up item I.

[11] "Frozen time" means that frame updates still occur, but time increments are 0. It's necessary to let generation go on, while dynamic processes, like physics and AI, are halted.

[12] Runtime part can ask, which objects should be placed in given bounding box of the universe, what is the geometry of these objects, etc.

[13] Because game scenario vector has been changed, or because of some editor command.

[14] E.g. mesh instances: characters, projectiles, etc.

[15] Generated resources are owned by generation part, and therefore are not affected by runtime restart.

single frame update, so that player wouldn't notice that it occured at all[16].

4 Editing

World level (Section 2.3) of generation can be edited interactively. Changes propagate through directed acyclic graph of dependencies, so that only affected parts are regenerated. For desktop computers, interactive editing can be implemented in two windows: the game (3D scene) and the editor (GUI).

Edited data is valuable, and must be preserved at all costs. It's desirable to minimize amount of code that can potentially lead to data corruption. One way to do this is to separate the game and the editor. The game is volatile, it should crash as early as possible to diagnose malfunctions. The editor should store its data after modification ASAP, to minimize loss in case if the editor crashes.

The editor consists of two parts: front-end and back-end. Front-end is essentially a GUI: JavaScript application running in web browser, that user interacts with. Front-end is communicating only with back-end. Back-end is responsible for data persistance and synchronizing with the game. Back-end communicates with the game and with front-end. It's preferable to have back-end as a standalone application[17]. This way data should be less likely to get lost or corrupted. The communication protocol between the game and editor back-end can be as simple as sending script code to execute on either part.

5 Scale

The game universe should be large. The challenge is that real-time computations[18] require using of floating point numbers, and these have finite precision. Only a small portion of the universe is immediately visible to the observer, though. Thus, one way to mitigate this challenge is to use "local" coordinate system, "centered" on the observer[19]. To represent observer's global position in the universe, another set of coordinates can be used[20].

Due to the limitations of floating-point precision of Z-buffering, there's an upper limit of the size of the scene, after which Z-fighting appears. To mitigate this problem one can render scene in multiple passes: *levels of detail*[21].

[16] Because already generated resources will be retrieved instantaneously after restart, without regeneration.

[17] E.g. using Nodejs.

[18] Such as affine transformations, physics simulation, etc.

[19] That is, all objects near observer will have coordinates near 0, providing full floating-point precision for computations.

[20] These coordinates specify observer's "offset" from the center of the universe. If we use double-precision floating point numbers to encode integer global coordinates in meters, this gives us a cube with a side of $\approx 2^{15}$ meters, which is $\approx 1,000$ times larger than Solar system.

[21] For example, render largest scene first (planets, stars). Then render smaller scene (landscape up to the horizon, mountains, clouds). Finally, render scene closest to the observer (grass, characters, buildings).

To navigate through the universe, one can "shift" between levels of detail. Only levels of detail larger or equal to the one the observer is currently in, are rendered[22]. Observer's movement speed is adjusted according to the current level of detail[23].

6 Grid

Only a small part of the universe is in the immediate vicinity of the observer in each moment of time. To split the universe into parts, one may use a *spatial grid*. Then, the query to the generation algorithm might look like: "generate the scene in local coordinates, corresponding to the part of the universe in a box with one corner in (x_1, y_1, z_1) and another in (x_2, y_2, z_2)". The algorithm can figure out how detailed the scene should be from the size of the query box.

7 Multiplayer

Gameplay should be hardcore, relying on player's skill rather than farming[24]. This requires immediate reaction to player's actions. Hot-seat multiplayer[25] fits well with this scheme. Distributed multiplayer in some way should be implemented as well. It's desirable to use peer-to-peer communication for multiplayer to avoid spending resources on servers. There are two modes of multiplayer interactions: active and passive.

In passive mode players just see each other's position[26] and chat with each other[27]. Passive mode should be able to function with any connection quality.

In active mode players actually play together as in hot-seat mode. Every player should see the same state of the game universe. One way to achieve this is to perform the same deterministic game state updates on each client: collect all players inputs, then update game state on every client using these same inputs. This scheme puts all players in the same advantage position: updates run as fast as slowest client does. Thus, it's desirable to play only with players having a good network connection. Players should be able to choose with whom they want to play actively[28], and to kick out players with slow connection.

[22] E.g. if the observer is on the level of "planets" and "stars", he shouldn't see "grass" and "buildings". Only "planets" and "stars" should be visible.

[23] Observer should move faster through the level of detail of planets, than he moves through the level of detail of grass.

[24] Like Mortal Kombat, Cave Story, Side Scroller, etc.

[25] Where all players are playing on the same system, sharing the screen.

[26] Like in Dark Souls: show a ghostly form of other players only. This approach saves the neccessity to synchronize game worlds between players, and softens the annoyance of remote players walking through things.

[27] Perhaps it's a good idea is to somehow utilize an established chat software/service, like IRC, Closed Circles, Facebook, etc. This same chat service can serve as an out-of-game mean of communication.

[28] E.g. by sending and accepting invitations.

8 Implementation

One of the requirements is portability and small platform-dependent code size. This means putting as much code as possible to the platform-independend part. Here is how these two parts may look like.

8.1 Platform-Dependent Part

The examples of tasks handled by this part are: window creation, render context creation, generating and rendering primitives (Section 2.1), working with network and filesystem, capturing user input, etc.

As all of these tasks are low-level, natural choice is to use native low-level language for the target platform. As of 2014, current bindings of OpenGL and OpenCL to scripting languages are somewhat limited and outdated. So in order to use latest available standards, best bet is to use a native language.

Possible choices are:

- In case of Windows or Linux it's C++.
- In case of Mac OS X or iOS it might be a mixture of C++ and Objective C.
- In case of Android OS it's Java.

8.2 Platform-Independent Part

The examples of tasks handled by this part are: concepts generation (Section 2.2), world generation (Section 2.3), handling user input, tracking global coordinates, interacting with the editor, scripts reloading, etc.

Most of the code will be in this part. Thus, the language must allow for writing compact programs, and also be fast enough, so that there's little incentive to rewrite this part to low-level language. The language must be highly portable and well-adopted in the industry as well.

Possible choices are: JavaScript[29], LuaJIT, C#.

8.3 Scripts Sandboxing

It might be possible to do scripts reloading entirely from Lua. *Host* Lua script will load *guest* scripts and run them in protected mode. If any of guest scripts throw error, host script intercepts it, shows notification, asks user to modify offending script and then reloads it. It's entirely possible to sandbox guest scripts from within host script by using `load` function with custom environment. It's also possible to setup a callback firing every N instructions using `debug.sethook`, which'd allow to break infinite loops.

[29] E.g. V8 implementation, as it sports JIT compilation. Although V8 isn't cross-platform, so portability might be an issue.

Remaining reasons to reload host script are:

1. Error in host script.
2. Out-of-memory.

9 Assorted Notes

1. Discussion, whether to use or not exceptions in C++ for game clients.
2. Think of a program execution in terms of possible histories of changes in the environment[30]. Validity of a program is a constraint on these possible histories[31].
3. For networking purposes, one can use ZeroMQ. It's small, flexible, mature, portable and well supported. There are bindings for Lua and NodeJs.
4. There's a way in NodeJs to monitor file changes in runtime: functions `fs.watch` and `fs.watchFile`.
5. Memory pools should always grow automatically and then report resulting sizes in the log. The goal is to avoid growing by adjusting initial sizes, but also to avoid crashing when these initial sizes are off the mark.
6. There's a code coverage tool for Lua, called LuaCov.
7. To verify if the game scenario is passable, one may use automated bots. Ideally, when bot is activated, at any point in the game, it controls player to complete the story line. Bot perceives current state of the game universe and emulates player controls[32]. To allow bot to perceive everything that it needs, it may be a good idea to design the game around the concept of bot from the very beginning.

[30] Environment can be a keyboard, display, memory, hard drive, etc.

[31] E.g. all possible histories for a game should contain states where each combination of keys can be pressed or released at any moment of time, while no game resource files are modified, and there's enough memory.

[32] E.g. keyboard, joystick signals, etc.