Animated Chatroom (rev. 2)

oldmud0

April 2019

(first rev. March 2018)

Contents

Preface

This preface is a personal account the events surrounding the formation and development of Attorney Online, the fan-made game that serves the impetus behind Animated Chatroom (which I will henceforth refer to as "AC").

In 2012, there existed a case-making site called Ace Attorney Online, (at the time, it was located at http://aceattorney.sparklin.org). The fan-made site grew to be successful: allowing players to create their own cases faithful to the style and flow of Ace Attorney, a plethora of such cases were made and could be played by anyone. It is certain that Ace Attorney Online attracted new players to the franchise; as such, legal recourse was never taken by Capcom.

In 2013, a Russian fellow named FanatSors decided to write a fan-made game of a similar intent, but with an entirely opposite execution. Dubbed "Attorney Online," the game was written in Delphi and did not support offline cases; rather, it encouraged the players to play out their own cases impromptu, following a looser format in comparison to the formulaic case format of Ace Attorney Online. In essence, AAO strived to be a simulator, whereas AO strived to be a chatroom.

In addition to YouTube videos recording the hilarity of case derailments in AO, Fanat further publicized his game by creating videos for "Turnabout Storm," a series which crossed over the *Phoenix Wright* saga with *Professor Layton* and *My Little Pony.* A moderate success, it was enough to bring the attention of my brother in 2015, approximately one year following the release of 1.7.5, the final version of Fanat's AO.

Personally, while I was not interested in the cases, as I had little time to sit through the drawn-out dance of roleplay, I was intrigued by the technical limitations of AO and sought to modify the game to add features, such as a way of automatically downloading assets instead of having to manually download them from every server by dumping a ZIP file onto a folder aptly named "base."

I invested a considerable amount of time in the summer of 2015 to attempt to reverse engineer the AO client. However, I knew little about reverse engineering, and I found that decompilation of Delphi executables to Delphi source was essentially impossible, as I had originally believed that Delphi used a bytecode (akin to Java). Desperate for source code, I asked wherever I could, yet no response was given. Burnt out from a fruitless project, I finally met stonedDiscord, a German fellow who wrote an open-source AO server (dubbed "serverD") written in PureBasic. As it turned out, he had already attempted to reverse engineer the AO client far before I even entered the scene, also having yielded few results from his efforts.

This caused me to look broader: what if the solution was to remake AO? It would be open-source, extensible, and contributable by anyone who wished to pitch in. But alas, like any of my other projects, it was simply too ambitious to ever consider being finished, especially if I was the sole contributor. With school and other, smaller projects in mind, I set AO to the corner of my mind and proceeded on with my life. Meanwhile, my brother grew bored of AO and moved the group that he had assembled from AO over to Roll20. (To this day, he still roleplays in Roll20 with around twenty other people in the group.)

About a year later, I decided to take a quick peek at state of AO again: how was this game doing? Was it dead yet? What kind of people were still playing? Surprisingly enough, I found not only a considerable number of people online, but also a remake being worked on by a Norweigan fellow named OmniTroid. Dubbed “Attorney Online Remake” (and later “AO2”), the project was being written in C++ using the Qt library. I thought little of the project, however, knowing the ramifications of such a massive undertaking. He is not going to finish, I thought. I scoffed a little and moved on with my life once more.

About six months later, I took another quick peek at this so-called remake, and taken aback, I found it at a formidable state. It was essentially complete.

I took interest almost immediately after this realization. I wanted to reach this OmniTroid person and tell them about my interest in working on the remake. In Feburary of 2017, in anticipation of the release of “AO2,” he established a Discord channel. I frequented it for the next few months, and after a few weeks of prying at the code and trying to make pull requests and issues wherever I could, Omni finally took attention of my eagerness to contribute to the project, granting me contributor access to the project and a developer role.

My first point on the agenda was a redesign of the asset system. After writing a design document regarding the subject and waiting a few days, he vehemently refuted my design, citing that it was too complicated, and the area was not important enough to address. I felt tempted to “be a rebel” and fork the code to implement the redesign myself, but before I even had the opportunity to start on the implementation (I was extremely busy at the time), Omni invited me to a private group with sD after asking me if I knew how to transfer a domain. It was July 8, 2017, a week after my surreal trip to Japan.

I did not really know what this secret meeting was about. It was clear that Omni desired to delegate to me control of more parts of the AO infrastructure (which he had established after the death of Fanat-controlled master servers and website), but why he decided to execute this *now* remained unclear.

I went through the migration process as he instructed. Service after service, nearly everything was handed over to me: website DNS records, the GitHub organization, the Discord server, and even the master server source code (which seemed to be hastily written in Python, and, to his admittance, was mostly held together by security-by-obscurity; to give you an idea, the script allocated one thread for each and every client).

At the conclusion of the migration process, I began to realize what was going on: Omni was leaving the project. A few minutes after I realized this, he said, “I’m sure you’ll figure it out. . . gl hf” and left the group. In a desperate attempt to confirm my prediction, I tried to contact OmniTroid privately, but no response came from him until much later. (When I was in Japan, I had been given an administrator role in the Discord server as well, which was around 200 members strong).

The next few weeks were stressful. sD and I had to rebuild the AO infrastructure, slipping off old components like Jenga blocks and replacing them with new ones, all while preventing any marked collapse, such as the website going down or the master server suddenly dropping. I took the next few weeks to write a new master server that would replace Omni's old code, racing against the clock to get it to working condition before he inevitably shut it down. In a stroke of luck, I brought the new master server to a working state and set it to standby a mere few days before Omni finally pulled the plug on the master server. The migration was fairly smooth, and an hour after receiving the first notifications of servers not appearing on the list due to the shutdown event, the new master server was set up, fixed a few times, and set as the default master server.

The conception of Animated Chatroom indeed began with the design of the replacement master server. The master server project had taken me longer than expected due to an intention to abstract the server to accommodate AO1, AO2, and future AO clients, which I expected would use Protobuf as a network protocol (since I vehemently despised Fanat's protocol and its use of ASCII characters to delimit packets).

By late August, I was already thinking of concepts and design that would drive Animated Chatroom, as well as reasons why AO2 simply could not be continued further. The source code simply was too messy to add new features to; moreover, there seemed to exist a number of difficult-to-trace memory leaks (a side effect of using C++ as if it were Java, I suppose).

In the beginning of my administration of the new AO community, I had prospects of merging with Visual Novel Online (VNO), a project developed by Fiercy, Cronnicossy, and FanatSors as an alleged "successor" to AO. It was also written in Delphi and likewise reverse engineered (albeit successfully) by sD. I relished the idea of a community that would use the same software for an all-inclusive variety of franchises; indeed, AO2 was being customized for role-playing games that had nothing to do with Ace Attorney, such as Danganronpa.

However, this idea of merging with VNO was not received well by anybody at all, due to my poor understanding of the composition of the VNO community. The two communities were wholly incompatible, and even Cronnicossy consoluted me in rejection of this idea. I quickly dropped it and continued my gaze on Animated Chatroom.

Sometime during my rediscovery of AO, I also learned of the so-called "AO 1.8," an unofficial update to 1.7.5 that made modifications to the default theme and vanilla assets. It included an "Automatic Attorney Utility," a robust set of macros for the game, since obviously the Delphi code could not be modified. I had briefly considered writing master server code to federate with the 1.8 master server, but I quickly abandoned the code after I realized that the 1.8 master servers were long dead. Mine was virtually the only one left standing.

During the fall semester, things were fairly uneventful. I was too busy to deal with Animated Chatroom, although I would go to the lab computers every Friday or so to work on some code or UI.

In December, during Christmas break, things predictably began to accelerate. What I did not predict, however, was an old friend's sudden interest in my Animated Chatroom project. Initially, he condemned me for my entire design and thought process being convoluted since the solution was "so simple," but after a few days of getting him up to speed and explaining why such and such was not a plausible solution due to a fundamental problem (following about the same line of thinking I had followed in the last six months), he finally was able to understand the justification behind Animated Chatroom. While I felt a tension between me and him due to his oversimplification of design objectives, we were nevertheless able to accomplish work on a Python client and C# server.

Following the break, however, things decelerated once more. The friend wished to place a deadline on "AO3," but I implored him that there would be absolutely no promises attached to this project, lest I break one and be humiliated eternally.

In February, my friend and another apparent developer had other plans: they wished to make an Electron-based client for AO3, and they would make it strictly for Attorney Online - it would not bear the name "Animated Chatroom." After a while of debate, I reluctantly agreed to help them, while denying that AC was cancelled. I wished to work on AC, but still could not find time.

Two months later, their AO3 project fizzled out. My inability to set apart a time to work on AC persisted throughout the rest of 2018, as I had been afflicted with particularly severe anxiety that summer.

By the beginning of 2019, I was able to better decide the optimal route to developing AC: instead of developing AC independently, it would be wiser to gradually morph AO2 into AC by stripping away its parts - a long-term strategy that could yield more results than one large ambitious project.

From the blurs of development, it has come to my attention that the design principles behind Animated Chatroom have not been unilaterally addressed. The purpose of this document is to serve as an up-to-date reference of the technical and ideological facets of the software. (After all, one must remember that the AO community focuses itself around little more than a piece of software; remove it, and there is hardly anything left that holds the community together.)

Notwithstanding the ultimate fate of AC, I write this document in hopes that it may serve useful for anyone who decides to take up this idea, although I personally tend to doubt it as well. Many parts of the design have undergone extensive debate; it would be wise to think of any problems before deviating from what has been written here.

Overview

Purpose

The purpose of Animated Chatroom is to allow players to use sprites to communicate with each other as if it were a visual novel, but without any specific objective or goal. The sprites can be from any game or creative work, and the program provides enough versatility to allow for complex situations to be played out.

Platform

The platform must be cross-platform and relatively slim. A bare install of AC should be less than 75 MB. The size of any required runtimes must also be factored into the size calculation.

Python

Python is a very friendly language and has proven to have a favorable learning curve for beginners and professionals alike. As a result of its reputation, many bindings and libraries exist for it, including PyQt5 and FFmpeg. However, it lacks a solid event-driven foundation needed for clean asynchronous programming.

Qt

Qt is a cross-platform desktop library that is predominantly developed with C++. However, it has bindings in various languages and can be statically compiled into a small single executable if needed. Breakthroughs in Qt 5.11 allow the execution of ECMAScript 6 code, which could potentially accelerate development since it is no longer necessary to struggle with C++.

Electron

Electron is a glorified wrapper for Chrome Embedded Framework (CEF), allowing OS calls while allowing the user interface to be written in standard HTML, CSS, and JavaScript. It is relatively recent in development, and its reputation remains to be established. Applications such as Atom and Discord use Electron.

If an Electron approach is taken, it is also advisable to keep a branch of the code for use in browsers (without the need of Electron).

Browser-based

Since the inception of HTML5, the browser has transformed from merely a layout engine to an entire application platform. As such, it is becoming increasingly easier to develop native-like applications for the browser.

Moreover, browsers come with WebM, GIF, APNG, Opus, and such resources can be loaded and played back very easily with JavaScript.

Some unsolved problems remain with this approach, however:

- Filesystem emulation using IndexedDB is not very well supported; as such, it may take more work to allow asset packages to be loaded in and extracted using browser-based JavaScript.
- The DOM is slow to update, and browsers generally do not guarantee timing. This means that the window can stutter at any moment.
- Browsers tend to be memory hogs; about 200-400 MB could be allocated for every open tab.
- Inter-tab/process communication is difficult to manage.
- Multi-clienting is difficult, since information is redundant across tabs.

If AC was less complicated, it could work on a browser. However, some features are desired by power users that cannot be replicated on a browser, such as certain hotkeys, as well as placing widgets across multiple monitors.

User interface

The user interface should follow this basic hierarchy:

- Each server occupies a main window.
- Each room within a server occupies a detachable tab.
- Each character occupied within a room occupies an in-character (IC) chat widget.

In addition to this hierarchy, almost every widget should be detachable from the main window, with the exception of the viewport.

Due to the convenient ability to edit UI via Qt Creator, it is very easy to customize the user interface. However, modders must be aware that additions to the UI in future versions may cause problems.

Lobby window

By default, this window should be shown on startup.

Server list (left pane)

There are two tabs: master server list and favorites list.

Servers are sorted by player count. Each item in the table contains the name, player count, protection level, and country of the server.

Servers can be easily added to favorites from the master list. Direct connect is also supported.

If a player selects a server already connected to, the respective window is focused and a "select room" dialog appears in anticipation of multiclient mode.

Server information (right pane)

The information on the server's item is repeated, although more verbosely. The description is also shown, as well as the designated URL of the server.

Options

Options are mostly implementation-specific. For the purposes of this documentation, however, I will refer to options that would be most appropriate for a Python-based implementation.

Moreover, an options window is an excellent way to introduce feature creep to a program. Options should never be introduced unless they have a legitimate use case and have been requested by a significant amount of users. For this reason, I will keep the options list short.

Name

The default name of the player.

Default asset repositories

The initial default asset repository should never be allowed to be deleted, in case of an accident.

Download custom assets by default

Some servers may allow players to bring their own assets into a server. By default, these assets will not be downloaded automatically because they may not be safe and can quickly accumulate in the local asset cache.

State the risks of allowing such automatic downloads when the “Yes” button is selected.

Scaling option

An illustration and extended description should be provided for each of the scaling modes.

Asset list

This is essentially a frontend to the database backing the asset cache.

There should be three options:

- forced downloading of an asset (add),
- removing an asset (delete), and
- pruning old assets not used in the last user-specified number of days (prune).

Loading dialog

The loading dialog should be modal to the lobby. A progress bar should be shown along with a description. See the handshake process in the network documentation for a detailed explanation of what may occur here.

Loading should be cancelable at any time during the process, although immediate cancellation is not guaranteed.

Asset downloads

An additional progress bar displaying the download progress for each individual asset should be shown, as well as the name of the asset being downloaded.

Main window

Join sequence

When a main window is opened, the lobby window should be closed if the origin was from the lobby. (It can then be reopened via `File -> Show Lobby` or some related sequence.)

The player is immediately asked to select a room. After a room is joined, the player is asked to select a character. If the process completes successfully, the main window's widgets then become accessible.

Room selection dialog

Each room is listed with a title, number of players, protection status, and description. Canceling this dialog when there is no active session causes a disconnect.

Character selection dialog

In the grid layout, only the character's icon is shown, and the character's name is shown on hover. If it is allowed, a button at the bottom named "Custom. . . " allows the player to search for a character in the asset cache.

Viewport

This is the central widget of the main window. As a result, it cannot be moved out of the main window. See the Core section for more information on how it works.

If a custom character that is not downloaded by the client is used in play, a star will appear at the top-right corner of the viewport. Upon clicking the star, information about the asset will be retrieved, and the client will be asked if they wish to download the asset.

In-character (IC) chat widget

This contains an adaptively resized chat box, as well as a grid view of available emotes, preanimations, and interjections for a character.

In the future, the chat box may provide auto-completion for the markup language stipulated in the Core section. It may also provide support for buffers in the future.

Chat should not be sent until the core reports to be idle.

IC log

This is simply a log of everything said in the IC log up to a certain point in time (e.g. last 5 minutes of server chat, chat since player joined, etc.)

Out-of-character (OOC) chat widget

This contains an adaptively resized chat box, as well as a bare-bones chat log.

Jukebox

This is a searchable tree view of the music assets available on the server.

Player list

This is a list of players currently in the room. If the server allows it, the character(s) associated with each player is also shown in a tree hierarchy.

If admin permissions are available, right-clicking a user or character will allow kicking.

Sound mixer

This provides a list of sound channels and faders/monitors for each of them. However, managing sound is troublesome due to the need for privilege management and such, so this widget is optional.

Server console

It may be possible to create a listen server from within AC, so a server console should be implemented. Moreover, it eliminates the need to recreate a separate server UI for dedicated server implementations.

Key-value list

This is simply a frontend to the settings list of the server.

Migration button

For listen servers only: a host may decide to hand over control of the server to another player. The host must wait until the player accepts or denies the request, as this is a modal dialog. See the Network section.

Assets

The asset system is loosely based on version 1 of the Docker container format, an uncomplicated format compared to subsequent versions. Although it seems ridiculous to base a small package format from a container format, I took this decision based on the necessity that differences between assets should be as small as possible. In many cases, modders tend to make derivatives of assets with very few differences between them, but where there exists a parent-child hierarchy, we can save space. This is basically the core idea that was taken from the Docker format; any other complexities were disregarded.

There are some large use case differences, however, that otherwise render the Docker format impractical for adoption as a model:

- Files within an asset do not need to be marked for deletion; they are simply left unused. (Besides, their method of tagging files for deletion seemed likely to collide with real files.)
- The tar format assumes that all files will be extracted in sequence; in reality, this is not the case. We may only need to seek for one file. Most implementations of tar assume a sequential read of the archive to find a file, even though the extended tar format includes an index.

Format

Every asset is uniquely identified in a consistent format. We shall define this format to be the SHA-256 of the archive file.

The archive file is a ZIP file that contains all information about the asset in a manifest file, along with accompanying data.

Manifest

The manifest file, named `info.json`, is a file that describes the asset and refers to files that are part of this asset or parent assets.

Chunks of a sample manifest are included below. The latest sample may be found in the fantaconvert repository. Fields starting with an underscore may be ignored, as they are purely informational (JSON does not allow comments).

Header

Required in all assets.

```json
{
  "name": "Adachi",
  "category": "character",
  "_category_options": [
    "audio",
    "background",
    "character",
    "evidence",
    "meta"
  ],
  "parent": "optional",
  "meta": {
    "author": {
      "name": "required if author field is included",
      "url": "optional"
    },
    "desc": "optional",
    "source": "[wikidata]",
    "date": "ISO-8601"
  }
}
```

Characters

```
{
  "chatbox_name": "optional",
  "side": "pro",
  "_side_options": [
    ["wit", "Witness"],
    ["def", "Defense"],
    ["pro", "Prosection"],
    ["jud", "Judge"],
    ["hld", "Helper defense"],
    ["hlp", "Helper prosecution"]
  ],
  "icon": "char_icon.png",
  "blip": "blip.wav",
  "_blip_comment": "extracted from base installation",
  "emotes": [
    {
      "name": "Coat On",
      "icon": "emotions/button1_on.png",
      "idle": "(a)normal.gif",
      "talking": "(b)normal.gif",
      "zoom": true,
      "_zoom_comment": "not present if false"
    }
  ],
  "preanims": {
    "coat": {
      "anim": "precoat.gif",
      "duration": 60,
      "sfx": {
        "file": "cool.wav",
        "delay": 60
      }
    }
  },
  "interjections": [
    {
      "name": "Hold it!",
      "sound": "holdit.wav",
      "anim": "holdit.gif"
    },
    {
      "name": "Objection!",
      "sound": "objection.wav",
      "anim": "objection.gif"
    },
    {
      "name": "Take that!",
      "sound": "takethat.wav",
      "anim": "takethat.gif"
    }
  ]
}
```

Packaging

The conversion of assets from the AO1 family (`char.ini` format) can be accomplished using the fantaconvert tool.

Large formats, such as WAV and GIF, should be compressed to the target/ideal asset format.

The resulting archive should be named either the unique ID of the asset, a truncated version of the unique ID, or a combination of the asset name and its unique ID. The actual name does not matter, so long as it cannot collide with other archives. The internal databases of the client and server will internally determine which assets correspond to which archives.

File formats

This section is still under debate. The proposed formats are as follows:

- **Still images**: PNG
- **Animations**: WebM/VP8 (YUVA420P), APNG
  We have determined that this is the only up-to-date format that supports 8-bit transparency with compression that matches or exceeds that of GIF.
- **Sounds**: MP3 (128~192k) or Opus (64~96k)
  MP3 remains the "gold standard" for music compression. However, recent audio sampling tests indicate that Opus provides nearly transparent quality at a low bitrate (96k). To save on size and as a trial adoption of the Opus format, Opus should be used for all new audio files. However, no transcoding need be performed if the file already exists in MP3 format, unless size is a significant concern.

Repositories

Asset repositories are essentially file servers hosting assets. They are decentralized; anyone can host an asset repository. Repository owners may impose any kind of standard they wish on assets submitted to such repositories. The minimum standard suggested is a similarity check, which is explained later.

Clients have a list of repositories they may download assets from. Clients download an index of assets from each repository and try the first repository that provides the asset. Servers may also offer a supplementary list of repositories that may be used in tandem with the client's local list. The local list always takes precedence over the server-provided list.

Standard repository index

```
["asset_id_1", "asset_id_2", ...]
```

Mini-repositories

In consideration of most server owners who wish to create private assets being unable to host entire repositories from their own networks, "mini-repositories" allow server owners to place a repository index on a file-hosting website. This repository index then lists assets and the URL to the ZIP file of each asset.

```
[{"id": "asset_id_1", "url": "https://dl.dropbox.com/s/..."}, ...]
```

The advantage to mini-repositories is that they can be stacked on a server repository list; they need not be reconstructed for every update. For instance, a server owner may generate a mini-repository for new ZIP files added in monthly updates, and the new mini-repository does not need to include the ZIP files from the previous months. The old mini-repositories are simply kept in the list.

Submission

Owners of public repositories should manually review submitted assets to control quality across the entire system. The reference implementation of the asset server should include a tool to thoroughly validate asset manifests to ensure no foul play has been done and that the asset does not already exist.

Due to the non-deterministic nature of the manifest (as a result of the inclusion of a timestamp, as well as whitespace and ordering differences), it is not possible to compare assets by a simple hash. Instead, the files within the archive must be compared by hash, and then the manifest must be compared by field, in order to generate a similarity percentage. (Since such comparisons must be performed against possibly hundreds of assets, a database or cache could accelerate the process.) If the highest similarity percentage calculated is greater than 85%, this indicates that the asset is probably a duplicate of another asset, and the parent of the submitted asset should be set to the asset which was duplicated. The submitted asset will have to be sent back to the submitter to make this correction.

After the manual verification, the repository owner may then place the asset archive in the server's storage and refresh the asset server.

Core

The core is the most interesting - and most challenging - piece of AC. There do not exist any “true” online visual novel engines, those which promote extensibility and abstraction.

Contrary to belief, it is more than simple animation, and it most definitely requires more work than simply strapping a video player onto the game. Timing *must* be considered as well as layering.

Attempted solution: Visual Novel VM

My first attempt to create an abstract core was to make a VM called the “Visual Novel VM,” and then allow the server to emit VNVM bytecode to clients. In this way, the work of the core would be eliminated from the client perspective: it would be the job of the server to translate messages into bytecode, which would get executed by clients. The state of the machine could be tracked perfectly, allowing near-perfect synchronization and trivial replay support using markers. I would also be able to get the headline for “write your next visual novel in assembly.”

A machine code concept is also employed in numerous released games. This explains the ease of porting some *Ace Attorney* games from 3DS to mobile.

Despite the purported convenience, the scheme was seen as impractical and complicated, so I decided not to pursue it any further.

Layers

Sprites can overlap each other with no restriction using layers.

It is the client’s final responsibility to determine what layers components of the scene will be placed on. A suggested configuration of layers is as follows.

- **Layer 0**: Background
- **Layer 1-2**: Character
- **Layer 3**: Auxiliary character
- **Layer 4**: Background overlay
- **Layer 5**: Effects

Only one sprite can occupy a layer at a given moment. (Otherwise, z-fighting would occur.)

Scaling

- The *base resolution* is the resolution of the lowest layer (the background).

- The *sprite resoluition* is the resolution of a sprite displayed on the scene. It may not be the same as the base resolution.
- The *viewport resolution* is the resolution of the viewport, as sized by the client. It may not be the same as either the base or the sprite resolution.

Two scaling methods are available:

- Sprites are scaled to the base resolution, and then the base resolution is scaled to the viewport resolution. The disadvantage to this method is that high-resolution sprites lose detail, but it allows all layers to be consistent in size.
- The base resolution and sprites are independently scaled to the viewport resolution. This is the naive solution, but it results in inconsistent layer quality.

Scaling filters include nearest-neighbor (point) filtering, bicubic filtering, xBRZ, and HQx. Nearest-neighbor filtering should be the default. It may be prohibitively complicated to upscale special media with other special filters, except if the upscaling is done as the last step by passing in the OpenGL framebuffer to the appropriate scaler.

Timescale

A server may define a timescale, a multiplier that controls the speed of preanimations, talking animations (debatable), postanimations, and dialogue. A timescale is also convenient for allowing slow clients to "catch up" with missed time due to a stutter or network lag.

Chatbox

As with all visual novels, a chatbox is necessary to show dialogue. The interval betweens chatbox ticks/blips, by default, is 80 ms, in which one character is printed out. The blip sound is defined by each character; there is no default blip sound.

Ticks should take consideration of special Unicode characters: some characters are diacritics, others symbols, and even some others unprintable. Diacritics and unprintable characters should be chained with the next standard character; symbols (such as kana and Han characters) should be printed at 0.75x timescale.

If a character's dialogue is blank, the chatbox should not be displayed.

The total time taken by a chatbox dialogue should be estimated to allow catch-up.

The appearance of the chatbox is defined by the scene as an asset. It is actually easier to go this route than to define it by the client's theme, since one would have to find a place to put these resources, and there is no better place than the asset cache.

The chatbox must may include a custom font. Some players prefer bitmap fonts, so appropriate support should be made for bitmap fonts by disabling subpixel smoothing.

A chatbox override should always be allowed for use by clients.

Backgrounds

Backgrounds are comprised of multiple sides, each being a static or animated sprite. Some sides may force a certain sprite direction (front, left, or right).

Backgrounds must specify a default side (for those characters which do not specify a default side), but this default side must not have a sprite direction restriction. When there is no activity, the default side should be shown without characters.

Characters

Characters are a series of related sprites bound together by a sequence or animation.

- **Name**: The name of a character will be shown on the character selection screen. A separate chatbox-friendly name can also be designated, which should be an abbreviated version of the name.
- **Side**: Specifies the default side of the background.
- **Icon**: The character icon will be shown on the character selection screen. It is possible, but not recommended, for a character to have no icon. The icon should be a minimum size of 64x64, although a size of 128x128 or 256x256 is recommended.
- **Blip**: Specifies the sound file to be played for each chatbox tick. If no sound is specified, no sound is played. The sound file must be contained in the asset.
- **Direction**: Specifies the direction the sprite faces.

Emotes

Emotes should be stored as an ordered list.

An emote consists of the following:

- User-friendly name
- Button icon

- Preanimation (optional). This can be prepended with a preanimation chosen by the user.
- Idle animation file
- Talking animation file (optional)
- Postanimation file (optional)
- Background override. Some emotes may require the use of speed lines in a certain direction. In this case, a background override will serve to be useful.

Preanimations

Preanimations need not be displayed in a certain order. They can be selected by the player, but if the same preanimation is selected as the one that is part of the emote, then the preanimation will only be played once.

A preanimation consists of the following:

- Name (key)
- Animation file
- Duration in milliseconds (optional)
- Sound effect (optional)
  - Sound file
  - Duration in milliseconds (optional)

The default duration of a preanimation is the duration of the animation.

Interjections

Interjections should be stored as an ordered list.

Interjections can interrupt any dialogue, except if the dialogue is protected.

An interjection consists of the following:

- Name
- Sound file
- Animation

The duration of an interjection is the duration of the sound effect.

Effects

Effects are an optional implementation feature. They allow, for instance, snow or some other post-processing effect to be overlaid on the scene.

Sounds

There are three types of sounds: sound effects, blips, and music. The volume for each sound type should be controllable by the client.

Sound effects

Sound effect files are located in the asset from which they are being played, although in the future they may also be included in “sound pack” assets. However, allowing standalone sound effects to be played is not an important feature, and thus sound pack assets should be disregarded.

Blips

See Chatbox section.

Music

Each track is an asset that can be named and categorized by the server. Since music tends to be very large, music should be either *downloaded on join* as a regular asset or *streamed* from an asset server. (This is under debate.)

It may also be the case that music is not an asset and instead streamed from a service such as YouTube. If this is so, servers should support adding non-asset tracks to their track lists.

Some servers may even allow any music at all to be played, even if it is not on the track list.

Between tracks, crossfade and fade out should be supported. In the case of crossfade, the new track should be loaded in before the crossfade begins.

Tracks loop automatically. Tracks stored as assets may contain loop start and end points; otherwise, the loop start and end points are defined as the start and end of the track.

The track selection privilege may be limited to a certain group of players. Such privilege management should be left to the server implementation.

Playlist support is not a feature currently under consideration.

Catch-up

When a client joins a room, the client should receive the last message sent and current track, both specified with an offset.

Custom events

Custom events, such as the "witness testimony" and "cross exanimation" animations found in Attorney Online, may be played. These events are managed solely by the server as overlays.

Markup language

A markup language may be considered to be used in IC for control of a character during dialogue. However, its design would complicate the core so much that it would rather not be implemented.

Messages

Instead of a "convoluted" virtual machine (see VNVM section), the core will be driven by a command set. This is different from an instruction set in that the state of the core is not tracked in detail; that is, there is no stack or explicit commands to load certain assets. The only communication with the core is comprised of a set of one-way commmands.

These messages should map closely with the network section.

`background <asset> [transition] [reset]`

Changes the background to the specified asset.

- `asset`: Asset ID
- `transition`: Whether or not a fade across black should occur
- `reset`: Whether or not the last emote/dialogue should be hidden. This will show the default side of the background with no dialogue or characters.

`emote <character> <side> <emote> [interjection] [preanimation] [dialogue] [offset]`

Plays an emote, beginning at the specified offset.

- `character`: Asset ID
- `side`: Side name of the current background
- `emote`: Ordinal number of the emote
- `interjection`: Ordinal number of the interjection
- `preanimation`: Name of the additional preanimation
- `dialogue`: Text to be displayed on the chatbox

- `offset`: Number of seconds of animation and dialogue to skip. If the offset is greater than the total estimated emote time, then the final state of the animation/dialogue is shown.

`timescale <scale>`

Sets the timescale.

- `timescale`: Multiplier

`music <asset/url> [transition] [offset]`

Changes the music (current track) to the specified resource.

- `asset/url`: Asset ID or URL. If this field is `stop`, then the music will be stopped.
- `transition`: Whether a crossfade or fade out should be done
- `offset`: The offset of the track to start from, in seconds. The current position of the track will need to be calculated by the client.
  - For music without loop points: `offset % length`
  - For music with loop points: `offset` if `offset < loop_start`; otherwise, `(offset - loop_start) % (loop_end - loop_start) + loop_start`

`event <asset> <animation> [interrupt]`

Animates a custom event.

- `asset`: Asset ID
- `animation`: Animation file
- `interrupt`: Whether or not dialogue will be interrupted as a result of this event

Network

The network protocol is what coalesces the client, server, and asset infrastrucutre into one unifying system.

Wrappers

For a desktop platform target, a raw TCP connection is suggested, with a 32-bit packet size preamble. For a browser target, WebSockets is recommended.

JSON

JSON is convenient for transmission across browsers; however, the format is somewhat verbose. It is intended to be transmitted via HTTP as opposed to raw TCP.

MessagePack

MessagePack is an adapted, machine-readable form of JSON that aims for more consistency over the wire. It is preferred if the initial implementation will be made using Python.

If MessagePack is used, the `bin` type must be used in order to allow Unicode strings to be parsed.

A field called `id` will be reserved for the name of the packet.

Protobuf and schema-based protocol libaries

Protobuf is a language that allows protocol schemas to be written. The largest problem with such schema-based protocols is their inflexibility: even a simple change in parameter order would break protocol compability.

NAT traversal

NAT traversal allows players to host servers without the necessity of forwarding ports. There are multiple candidate solutions of making this possible:

- **STUN**: Session Traversal Utilities for NAT is an IETF-standard method of allowing two clients to establish a direct connection with each other behind various NAT types. Google offers a STUN server; however, STUN works best with UDP, and some STUN server implementations do not work well with TCP.

- **UPnP**: Many routers support UPnP, which allows port forwarding programmatically without the necessity of configuring ports via a router configuration page. However, based on experience, UPnP support has been shown to be flimsy and unreliable.
- **Samy Kamkar's pwnat**: May be construed to be malicious, but claimed to be extremely reliable.
- **localtunnel**: A simple public tunnel server.
- **IPv6 over Teredo**: Windows attempts to initiate a Teredo interface when a connection is attempted to be established to an IPv6 host. This results in the client receiving an IPv6 address and exposure of all ports via this address. However, Teredo support is unreliable, as it is disabled by default for most Windows machines.
- **VPS**: Ask players to pay money to host their own servers via an automated service. Extremely inconvenient.

Master server

The master server simply holds a list of IP addresses and sends them to the client.

The master server may be implemented using any high-performance network library, such as Python `asyncio` (`uvloop` if more performance is needed), Node.js (if project is time-constrained), or Elixir (if competent enough to pursue it).

The server list may be backed using Redis or PostgreSQL.

For each server on the list, the client will attempt to connect to each one and retrieve their basic information.

`servers`

Client to MS

Gets the server list and subscribes to changes.

Response: `servers <servers>`

`servers <servers>`

MS to client

Represents the full list of servers.

- `servers`: List of servers
  - `ip`: Address of the server; may be IPv4, IPv6, or hostname
  - `port`: Port of the server

`new-server <ip>`

MS to client

Indicates that a new server has been added to the list.

`drop-server <ip>`

MS to client

Indicates that a server has been removed from the list.

`heartbeat <port>`

Server to MS

Requests that the server located at the connected IP address and specified port be added to the server list for some amount of time.

Response: `heartbeat-<ok|err>`

`heartbeat-ok <interval>`

MS to server

States a success in the heartbeat request. The next heartbeat request should be sent within `interval` amount of seconds.

`heartbeat-err [message]`

MS to server

States an error in the heartbeat request, with an optional short error.

Handshake

Overview

- Client establishes a connection. (TCP SYN)
- Server waits for any client message (~5 seconds).
- The server should not send server status to any connecting client, as it is a large waste.
- Client sends `info-basic`.
- Server sends back `info-basic` response.
- Client sends `join-server`.

- Server sends `join-server` response.
- The client should proceed to request assets from the server.

`info-basic`

Client to server

Requests the basic information of the server.

Response: `info-basic <...>`

`info-basic <...>`

Server to client

Provides the server's basic information to the client.

- `name`: Name of the server
- `version`: Server implementation version
- `player_count`: Number of players currently connected
- `max_players`: Maximum number of players
- `protection`: Protection level of the server:
  - `open`: No password is needed to join at least one room.
  - `password`: Joining the server itself requires a password.
  - `spectate`: Joining the server is restricted, but spectating is open. (Password/whitelist access may be needed to join the server.)
  - `closed`: Server is enforcing a whitelist, or it is not open to any new players.
- `desc`: Server description
- `auth_challenge`: Authentication challenge. Passed even when there is no password.
- `rooms`: An ordered list of rooms
  - `id`: Unique identifier for the room
  - `name`: Name of the room
  - `players`: Number of players in the room
  - `desc`: Description of the room
  - `protection`: Protection level of the room
    - `open`: No password is needed to join at least one room.
    - `spectate`: Access to the room is restricted, but spectating is open.
    - `closed`: The room is whitelisted or closed to all players.

`join-server <name> <auth_response>`

Client to server

Indicates that the player is ready to join the server.

- `name`: Name of the player
- `auth_response`: hmac-sha256(auth_challenge, password) Required even when there is no password.

Response: `join-server <result> [message]`

`join-server <result> [message]`

Server to client

Indicates the result of the join request.

- `result`: Result code
  - `success`: Client joined successfully.
  - `full`: The server is full.
  - `password`: The password is incorrect.
  - `banned`: The client is banned.
  - `other`: Requires a message.
- `message`: Custom, optional server message

`disconnect [message]`

Server to client

Indicates that the client was disconnected by the server due to some reason. A TCP RST follows after this message.

- `message`: Custom, optional server message

Assets

`asset-list`

Client to server

Requests the full asset list of the server.

Response: `asset-list <repositories> <assets>`

`asset-list <repositories> <assets>`

Server to client

Lists the server-suggested repositories and full asset list of the server.

- `repositories`: A list of server-suggested repository URLs.
- `assets`: A list of asset IDs.

`new-assets <assets>`

Server to client

Indicates that new assets have been added to the server for the client to download in-game.

NOTE: This message may be received while the client is still downloading assets as a result of an `asset-list` response. Any assets added from the `new-assets` list must be added to the download queue if they do not already exist.

- `assets`: A list of asset IDs.

Rooms

A client session includes only one room. Joining another room will require a new client session/socket to be opened with the server.

`chars <room_id>`

Client to server

Requests the character list for a room.

- `room_id`: Unique ID of the room

Response: `chars <room_id> <characters> <custom_allowed>`

`chars <characters> <custom_allowed>`

Server to client

Lists the allowed characters for a room.

- `room_id`: Unique ID of the room
- `characters`: Ordered list of characters
  - `asset`: Asset ID
  - `protection`: Protection level
    * `open`: Character can be used
    * `used`: Character is in use
    * `protected`: Character is protected
- `custom_allowed`: Whether or not custom characters not on the list are allowed

`join-room <room_id> [character]`

Client to server

Joins a room.

- `room_id`: Unique ID of the room
- `character`: Asset ID

Response: `join-room <result>`

`join-room <result>`

Server to client

Indicates the result of the room join response.

In addition to this, the server should separately send events to allow the client to catch up to last activity.

- `result`: Result code
  - `success`: Client joined successfully.
  - `full`: The room is full.
  - `denied`: Access is denied.

`ooc <message> [player]`

Bidirectional

Sends an OOC message to the server, or receives an OOC from the server or another player.

- `message`: The text content of the OOC message.
- `player`: The originating player of the message. Ignored if sending from client to server. If the OOC message is a server/system message, this field will be empty.

Response: OOC message is echoed by server.

`time`

Client to server

Requests the network time for the client to synchronize with the server.

Response: `time <time>`

`time <time>`

Server to client

Indicates the current network time of the server.

- `time`: Milliseconds since epoch (can be any epoch)

Events

The server will send event commands that may be passed directly to the core; see the Core section for information about such commands.

In addition, events will contain a field `timestamp`, which specifies a timestamp representing when the event should be fired by the client. To synchronize clients, a slight (25 ms) delay should be added by the server to all scheduled events.

Many commands are bidirectional; discretion should be taken in the server implementation regarding which commands should be bidirectional. Invalid commands sent by the client are ignored.

Administration

Privileged users may access these commands.

`opts`

Client to server

Requests the list of options offered by the server.

Response: `opts <options>`

`opts <options>`

Server to client

Lists a key-value map of options offered by the server. Values may contain nested keys and values.

- `options`: Key-value map of options

`set-opt <key> <value>`

Client to server

Sets an option. The value may not be an object, but it can be an array or literal value.

- `key`: Dot-delimited key
- `value`: JSON-like value

Response: `set-opt <key <result> [message]`

`set-opt <key <result> [message]`

Server to client

Indicates the result of the most recent option change.

- `key`: Key attempted to be set
- `result`: Result code
  - `success`: The option was set successfully.
  - `error`: There was an error setting the option. The error should be assumed to be that access was denied.
- `message`: Optional error message

Migration

This is an optional feature.

`migrate <config>`

Server to client

Sends an offer to migrate an entire server to a client, assuming that the client has listen server capabilities.

- `config`: The entire configuration key-value store

Response: `migrate <result>`

`migrate <result>`

Client to server

- `result`: Result code
  - `success`: The client accepts the migration, and the old server should shut down.
  - `error`: The client rejects the migration.

`redirect <ip> <port>`

Server to client

Broadcast to all clients that the server is being moved to a specified address and port.

- `ip`: New address of the server
- `port`: New port of the server

Implementation

The design of Animated Chatroom may seem long, complicated, and irrelevant. For this reason, an incremental development process is recommended.

A suggested roadmap is as follows.

Initial implementation set

- Implement core, except for the following features:
  - WebM support
  - Custom scaling filters
  - Duration estimation (i.e. ignore offset)
  - Custom chatbox fonts
  - Effects
  - Transitions
  - Audio fade
- Implement server, except for the following features:
  - Protection
  - Permissions
  - Idle activity event
- Implement the following features in the client:
  - SQLite-based asset manager
  - Lobby
    - Master server list
    - Direct connect
    - Join dialog, including asset download progress
  - Options
    - Name
    - Repository list
    - Asset list
  - Main window
    - Room selection dialog
    - Character selection dialog
    - Viewport
    - IC widget
    - IC log
    - OOC widget
    - Music list (no categories)
    - Client-side sound mixer
    - Basic server configuration dialog
      - JSON object tree
- Converter from AO1 to AC assets

Subsequent features

Since there will be many bugs and features to be added, this section intentionally remains blank. The developers' best judgment should be applied, given they have even reached this point in the development process. Bugs relating to the initial implementation set should be prioritized.

Best of luck to all future AC developers!