SAGS v1.1 Technical Specification

Please check out Spatial Audio Guidance System source code in the Github Repository:

https://github.com/DanielHou315/Spatial-Audio-Guidance-System

Hardware

Stereolabs ZED 2 Camera

The Stereolabs ZED 2 stereo camera is chosen for this project because the ZED SDK integrates SLAM, object recognition, and object tracking into one package that is simple to use and has plenty of room for expansion.

The main advantage of the ZED 2 camera over Intel's RealSense series is that it is able to detect object within up to 40 meters, multiple times that of Intel's maximum detection range.

The primary downside of the ZED 2 is that it does not use ToF or RGBD to calculate depth of each pixel, and therefore will be unusable in low lighting conditions.

Nvidia Jetson Xavier NX

The Nvidia Jetson Xavier NX developer board is chosen for this project primarily because of its CUDA-enabled GPU. Since some features of ZED SDK requires CUDA to run, Xavier NX is the most compact and least power hungry choice for outdoor testing. Its capable CPU and desktop Linux environment are also suitable for the development of this program.

Battery Pack

Any battery pack with 110v output will be able to power the Jetson Xavier NX board to function in an outdoor environment. The specific battery I used is a FlashFish E200 sponsored by the Haverford School Science Department.

Headphone

Any dual or multi-channel headphones will work for this device (so that users can experience the effect of spatial audio). In the development of SAGS v1.1, both a Beats Solo 3 and an AirPods Pro have been used.

Development Environment

SAGS is developed on Jetpack 4.5 for Jetson Xavier NX with CUDA 10.2.

The object recognition, tracking, and positioning are based on ZED SDK 3.5 for Jetpack.

Python scripts are executed in Python 3.6.

For easy configuration of ZED SDK, I did not setup Docker or other containers for this development.

Software

Object Recognition

Object Recognition in SAGS is achieved through ZED SDK's Object Detection API:

https://www.stereolabs.com/docs/object-detection/using-object-detection

All Object Recognition related parameters in SAGS are defined in main.py for camera initialization.

I have experimented with YOLOv4 integration with ZED, but while it detects a wider range of objects, YOLOv4 is much slower than ZED's in-house module (5 FPS vs 25 FPS in medium intensive scenes). This is why ZED SDK is the eventual choice for SAGS object recognition.

Google's MobileNet is a promising object recognition algorithm for mobile platforms, and SAGS integration with MobileNet may come in a future update.

Object Positioning

Object Positioning in SAGS is also achieved through ZED SDK's Object Detection API. It is based on ZED 2's SLAM and Object Detection features.

For SLAM, ZED SDK will take images from both of the cameras, compare the images, and calculate the depth of each pixel with a built-in algorithm that also considers IMU data and previously generated models to construct a 3D map of the surrounding environment.

ZED SDK's Object Detection API returns various information about an object it detects. The ones I use in SAGS are:

1. Object ID (unique to each object detected throughout the execution of the program)

2. Object type and sub-type (Person, Vehicle, Animal, etc)
3. Object Position [x_position, y_position, z_position], relative to the left sensor of the ZED camera.
4. Object Velocity [x_velocity, y_velocity, z_velocity], relative to the left sensor of the ZED camera.

All of these information are loaded to the spatial-audio module for spatial audio rendering as properties to a **Source**.

Spatial Audio Rendering

Spatial Audio Rendering in SAGS is based on OpenAL.

i. A currently maintained fork of OpenAL project is OpenAL-soft: https://openal-soft.org/
ii. Basic OpenAL structures of SAGS is based on the work of Youtuber Code, Tech, and Tutorials: https://www.youtube.com/channel/UC4EJN2OSNdl-mSxGjitRvyA
iii. Custom management functions and structures are added to support object management.

All Files related to Audio Rendering is in the audio_module directory.

Audio Context

The Audio Context includes everything that OpenAL needs in order to function. The Audio Context setup is completed at the start of the program and destroyed upon ending of the program.

Buffer

The Buffer is where all source audio files are loaded. During SAGS setup, the program loads all audio files it needs into a buffer, so that the renderer can refer to these files during real-time rendering. In SAGS, each audio effect represent a certain type of object. The objects able to be detected and audio-rendered in SAGS v1.1 are the following:

1. person
2. bike/motorcycle
3. car
4. bus/truck
5. cat
6. dog

All source audio files are mono-channel, 16bit, and stored in sound_source/buffer directory.

Source

A source in OpenAL is equivalent to one object in the physical space or one source where the sound comes from. In SAGS, a source has the same properties as an object detected by the Object Detection module in ZED SDK.

When an object is detected by ZED SDK, whether for the first time or not, the program creates/updates its information to OpenAL with this function in video_module/AudioInterface.py:

```
# If the ID is not registered in OpenAL, register the object first
# Record the presence of this object in the current frame
# Update object Type
# Update Object velocity
def update_source(id, type, pos, vel, state):
    if audio_lib.is_new_source(id):
        audio_lib.create_source(ctypes.c_int(id),ctypes.c_int(type))
    audio_lib.add_current_source(ctypes.c_int(id))
    audio_lib.update_source_position(ctypes.c_int(id),ctypes.c_float(pos[0]),ctypes.c_float(pos[1]),ctypes.c_float(pos[2]))
    audio_lib.update_source_velocity(ctypes.c_int(id),ctypes.c_float(vel[0]),ctypes.c_float(vel[1]),ctypes.c_float(vel[2]))
```

All sources (equivalent to all objects detected in a certain frame) are updated with calculated position and velocity values so that their behaviors match those objects they represent in the real world.

i. If the real-world object moves to the right at 5 m/s, OpenAL renders the sound as if the speaker playing the sound is moving to the right at 5 m/s.

Source Management

For clutter-recycling, if there is no object being detected, SAGS is designed to kill all rendering and delete all established sources to indicate that there are no object detected in frame.

If there are objects detected, the audio management creates a source and its linking mechanisms:

```
/*
If the ZED ID is not currently registered in OpenAL:
1. Push the ZED-detected object ID (let’s call it ZED ID) into a vector previous_source defined in AudioModule.cpp.
2. Link the ZED ID to a Unique ID (an unsigned integer starting from 0 and increasing by 1 each time a new source is created) for this spec
3. Link the Unique ID of this object to the OpenAL Source pointer in a map structure.
4. Play the sound and set it to looping.

Otherwise, return error: object registered
*/

void create_source(int ID, int type){
        if (is_new_source(ID)){
            previous_source.push_back(ID);
            Source_ID_map[ID] = number_of_IDs_registered;
            Source_map[number_of_IDs_registered] = new SoundSource();
            number_of_IDs_registered++;
            Source_map[Source_ID_map[ID]]->Play(Buffer_map[type]);
            Source_map[Source_ID_map[ID]]->SetLooping(true);
        }
        else{std::cout << "There is an error creating source: source ID already created" << std::endl;}
        return;
    }
```

Through this, even if future implementations of other Object Detection algorithms recycle object labels, OpenAL will be able to manage each source with a unique ID throughout the execution of the program.

Likewise, when a source is deleted:

```
/*
If the object has been loaded to OpenAL before:
1. Its OpenAL source object is deleted.
2. The map from Unique ID to OpenAL source is erased.
3. The map from ZED ID to Unique ID is erased.
4. ZED ID is removed from the vector.

Otherwise, return error: object not registered
*/
void delete_source(int ID){
        if (!is_new_source(ID)){
            Source_map[Source_ID_map[ID]]->Stop();
            Source_map[Source_ID_map[ID]]->~SoundSource();
            Source_map.erase(Source_ID_map[ID]);
            Source_ID_map.erase(ID);
            for(int i = 0;i < previous_source.size();i++){
                if(previous_source[i] == ID){
                    previous_source.erase(previous_source.begin()+i);
                    return;
                }
            }
        }
        else{std::cout << "This ID is not REGISTERED so not DELETED!!!" << std::endl;}
        return;
    }
```

As a result, memory usage of this program remains in reasonable levels.

i. It left plenty free with the 8GB memory on my Xavier NX board after an hour of street testing.

ii. Memory usage is accessed through jetson_stats: https://github.com/rbonghi/jetson_stats

A separate vector current_source in AudioModule.cpp records the ZED IDs for object detected in the current frame. When every object detected is processed and loaded into OpenAL, the program compares this data with previous_source to see if there are objects that were detected in the last frame but are not detected in this frame. The program then deletes these sources to indicate that they are no longer in frame.

Due to ZED SDK's retention feature, even if an object goes out of frame for 3 seconds, the program is able to recognize it when it returns into frame. During this time, ZED SDK continues to feed predicted object information to OpenAL and the sound is rendered continuously, even if the object is out of the picture.

Object Filtering in Audio Rendering

For the purpose of the demonstration, I did not implement any object filtering strategies while making the maker portfolio/demonstration video. This means that all object detected in a frame will be rendered into spatial audio.

While looking at 10 cars may not be a huge burden for our brain, listening to 10 cars running at the same time while trying to distinguish their positions can be troubling. It is for this reason I explored strategies for object filtering—to choose a few objects to render, rather than to render sounds for all objects, in complex scenes.

Based on the principle of continuity and risk-assessment, I implemented my first version of filtering algorithm that allows up to 3 object rendered at the same time:

1. I prioritize already-rendered objects to continue rendering until they go out of frame.
2. Then, I calculate the minimum distance other objects will be from the user with its velocity data and point-line minimum distance formula.
3. I calculate where minimum distance occurs and calculate the time it takes the object to reach that point.
4. If an object's minimum distance is within a threshold (considered "risky" for user), they are rendered to let the user know. Otherwise, sources are not created for the other objects (that are "safe") in OpenAL.
5. Continue the logic in the next frame.

Sadly, since ZED returns velocity data that are estimated, and therefore, fluctuating and inaccurate, this algorithms often miscalculates risks of objects and does not function well in the real world.

Still, developers will be able to access the minimum distance or the time it takes for an object to reach that point with the function

```
def outdoor_filter(obj)
```

in video_module/ObjectFilter.py.

Human-Machine Interaction

To help users and developers better diagnose the state of this prototype software, SAGS uses Playsound module in Python to give audio notification on the state of the system:

```
import Playsound
```

All of the audio files for SAGS human-machine interaction are stored in the sound_source/procedure directory.

The human-machine interaction features in SAGS include the following:

1. On startup, **startup.wav** will be played to let the user know that the program starts.
2. The main function configures OpenAL, the audio module, first. If OpenAL setup fails, **audio_fail.wav** will be played.

3. The main function then configures ZED 2 stereo camera. If the camera fails to start, **camera_fail.wav** will be played.
4. If Both audio and video is setup properly, **prepared.wav** will be played to let the user know that the program is ready for object tracking and spatial audio rendering. It typically takes two seconds after finishing this audio file playback to let object tracking really start working.
5. Upon closing the program, user inputs Ctrl-C on the keyboard, and after the main function completely closed everything else, it will play **shutdown.wav** to let the user know that the program is closed entirely.
6. Both Playsound and OpenAL defaults to system defaulted audio device, so if users do not hear any audio being played, it is likely that they are using a different audio device than system default. SAGS cannot change system default audio device yet and requires manual change.

There are text messages displayed in terminal for all of the above human-machine interactions for debugging, but users will be able to figure out what failed with just the audio played by the main function.