WebAssembly

IJCLab dev

9 Février 2021

Guy Barrand, CNRS/IN2P3/IJCLab

WebAssembly

- Une idée, technologie intéressante pour passer sur le Web une application en C++, qui fait du WebGL pour son graphique, et ce sans avoir à faire et déployer un serveur spécial.

Le secret

Dans vos navigateurs de toile, il y a une machine virtuelle ! (si, si) : la wasm.

(une « portable virtual stack machine », dixit Wikipedia)

Comment ça marche ?

- Dans vos navigateurs de toile, il y a une machine virtuelle !
- On installe sur son mac adoré, ou son Linux bien-aimé, ou son Windows abhorré, la boite à outil « emsdk ». (em est pour « emscripten ». (Aucune idée de ce que ça veut dire)).
- On cross-compile son appli avec « em++ » pour fabriquer un « .wasm » (binaire), quelques .js et un index.html. (em++ utilise clang et LLVM).
- On déploie {index.html, .wasm, .js} dans des pages statiques chez n'importe quel hébergeur (par exemple gbarrand.github.io pour moi).
- Donc pas besoin de déployer un serveur spécial.

Comment ça marche (2) ?

- Lorsqu‘on charge le index.html depuis son navigateur, le .wasm est chargé et exécuté dans la machine virtuelle, donc sur votre machine en local.

ET VOILA !

Graphique?

- La base est de faire du WebGL. Donc on a du 3D.
- Il y a une pauvre implémentation de GL-ES-1 sur WebGL dans la boite à outil emsdk, mais c'est à éviter (très incomplet).
- WebGL : depuis le C++ on fait donc de la "string programming" = on fabrique du code javascript qui est passé au navigateur de toile pour compilation et exécution, et qui lui même va finir par faire de l'OpenGL compilé.
- Donc on a forcément une couche d'inefficacité comparé à la même appli compilée en local et faisant en direct de l'OpenGL compilé.

Mes applis

- C++98 : ok avec em++. Ce que j'utilise de STL/STD passe.
- Ma logique de graphes de scène passe.
- Mon code pour lire des .root passe.
- maitrise des externals : j'embarque, depuis toujours, le code de mes externals critiques : freetype, expat, zlib, jpeg, png. (Il n'y a rien de tout cela venant avec le SDK de emscripten !). Et cela passe aussi.
- J'ai fait le portage de cfitsio, hdf5. Geant4 core passe !
- Je peux faire tourner mes applis ! ☺☺☺
- En particulier du fait que je fais mon interface utilisateur avec ma logique de graphes de scène et donc en WebGL (graphique unifié).

À savoir

- C’est en 32 bits.
- L’utilisation des sockets est bloquée. Pour faire du http il faut passer par le navigateur de toile que va faire des requêtes en asynchrone: ça complique.
- On peut uploader un fichier de données de la machine locale à la machine wasm. On peut aussi récupérer un fichier produit sur la machine wasm sur la machine locale.
- On peut encapsuler des fichiers de données dans un conteneur .data vu par l’API C standard (fopen, etc…)
- Le windowing est fait dans un canvas et peut s’insérer dans une page web quelconque. En particulier on peut mixer avec du GUI fait en javascript.

Physique : HEP

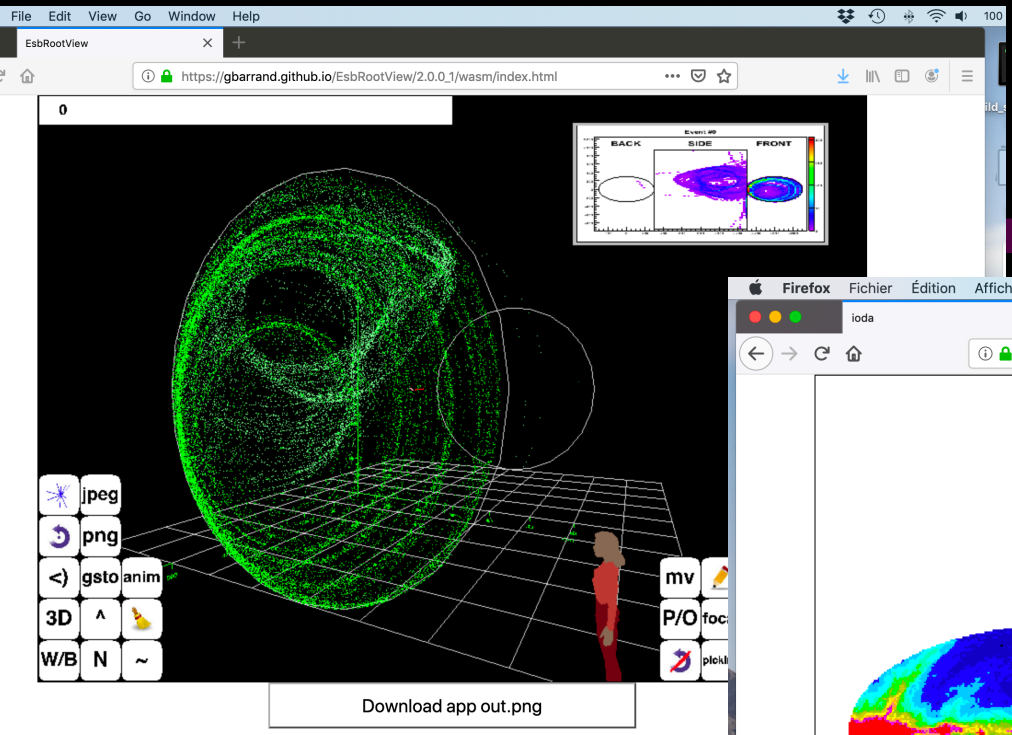
- J‘ai pu passer g4view, g4exa : et donc avoir Geant4 en .wasm.
- Je peux lire des fichiers .root de géométrie ou d‘analyse.
- pmx marche : donc un embryon d’event display d’LHCb (pour montrer, encore et toujours (soupir), à HEP et au CERN qu’on peut faire du soft potable portable).
- EsbRootView : display pour de la R&D pour ESSnuSB (ESS accélérateur à Lund). (Une suite inattendue d’une visu Geant4 MEMPHYS qu’on avait fait avec Jean-Eric Campagne il y a… une vingtaine d’année).
- Tout cela est essayable depuis gbarrand.github.io, depuis les sections « WebAssembly » de mes applications.

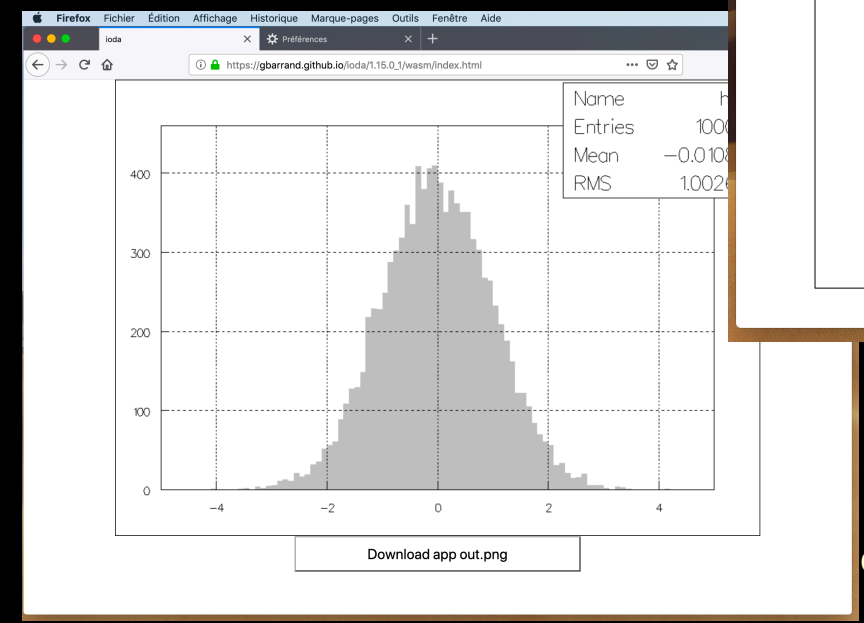
Physique : analyse, astro

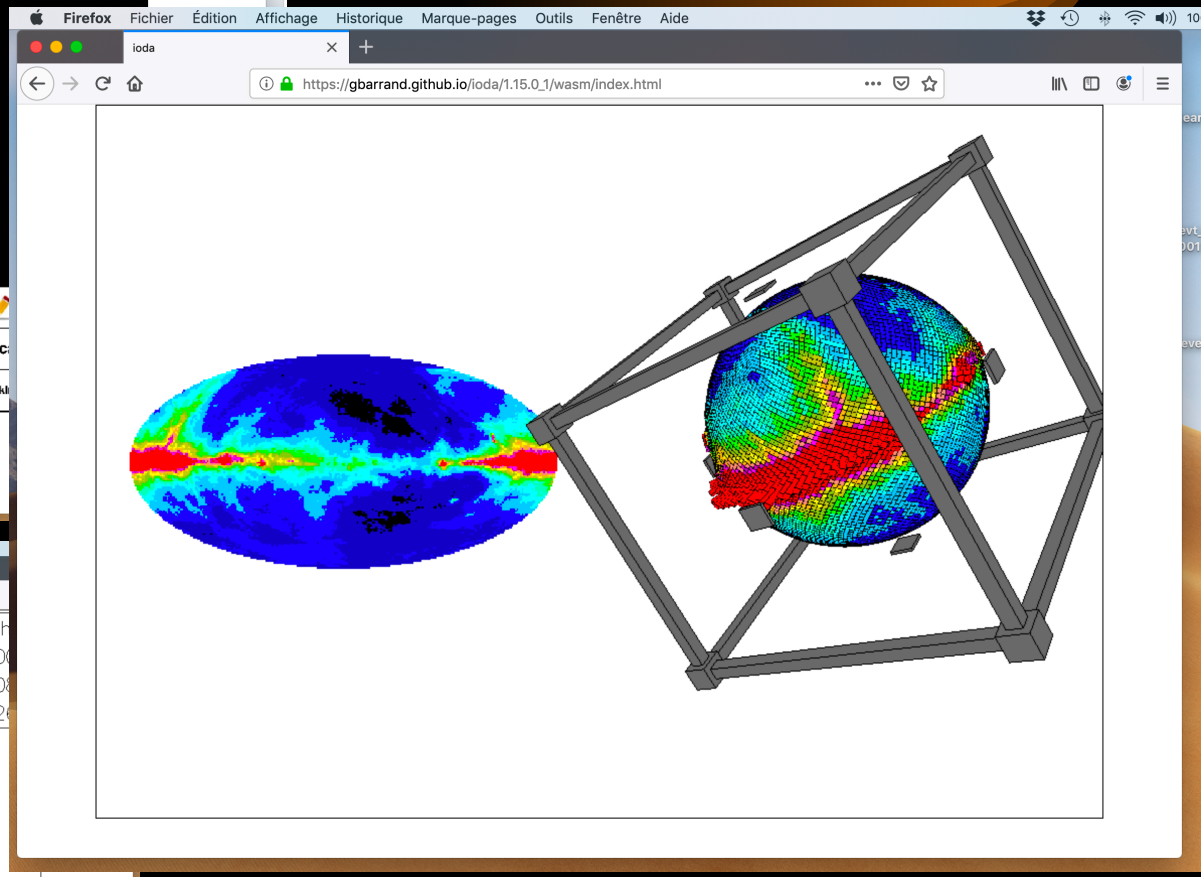
- ioda fonctionne : et donc mon plotting, une partie de SophyaLib, cfitsio, hdf5, lua et mon lecteur de fichier .root.
- Une partie de TouchSky : et donc une visu de HEALPIX et de projection mollweid.
- MAIS, du fait de limitations de emsdk sur les sockets, je ne peux pas faire des requêtes en synchrone sur STSCI ou le CDS. Il a fallu repenser, depuis le C++, les requêtes http pour les passer par le navigateur de toile en asynchrone (ça complique le design de l'appli).

Ça marche…

- Ça marche pour moi avec Firefox et Chrome sur mon Mac (donc adoré), sur iOS et Android, sur Windows-10 et Firefox sur des VM Linux (centos7 et MINT).
- MAIS, marche mal sur Safari (qui impose une limite sur la mémoire utilisée). Ok avec des petites applis.
- ATTENTION, WARNING, ACHTUNG : {.wasm, données liées à l'appli} peut être gros, donc le chargement au démarrage peut être lent si on a un internet pourrave pour accéder au site de distribution.
- (Mais bientôt on sera tous fibrés avec de la 5G :-))







d, CNRS/IN2P3/IJCLab

Wasm et la tour de Ba(by)bel

- Visiblement on peut cross-compiler avec Go(Sport), C#, java et Python.
- Ce qui rajout des couches…
- (Enfin, moi ce que j'en dis…)

- Une certaine similarité avec docker. Graphique : avec docker on est condamné à X11/Mesa, avec "wasm" c'est… moins pire. Avec docker on a un vrai Linux. Avec wasm, distribution moins contraignante, docker induit une machinerie qui prend beaucoup de ressources en local.

WebAssembly ☺☹

- Enfin une connexion « C++ WebGL » avec le Web intéressante !
- Grâce à mes choix stratégiques mes applis passent.
- MAIS on sent bien, à l'usage, que les navigateurs de toile sont clairement « pensés » pour exécuter des « petites tâches de manière asynchrones », et pas pour tourner des grosses tâches synchrones.
- Certaines navigateurs bloquent ces tâches.
- De toute façon le graphique, est moins réactif qu'en « pur local ».
- Mon sentiment est que le wasm sera très bien pour de l'outreach ou des tâches de physique très ciblées et n'embarquant « que ce qu'il faut ». Et qu'on doit encore cibler de tourner en local pour les « grosses applis » (bref, rester proche du silicium).

Demos…