Déploiement de TP de simulation avec Dev Container

On suppose qu'on veut déployer des TP qui font appel à des environnements de simulation, comme Gazebo ou URSim.

Pour faciliter le déploiement reproductible en général, et en particulier la réutilisation d'environnements de développement fournis par les développeurs, on utilise des conteneurs Docker. On a besoin d'accélération graphique via un GPU pour afficher ces simulations afin de ne pas surcharger le CPU.

Installation de Docker

On peut faire tourner le conteneur Docker sur n'importe quel environnement : Linux, Windows, Mac, ou sur un serveur

Développement local sur Linux avec X11 et pas Wayland

- Installer par exemple Ubuntu 24 ou Linux Mint 22 Mate
- Vérifier que le système de fenêtrage est X11 et pas Wayland
- Installer docker.io : le paquet debian de docker
  `sudo apt install -y docker.io --install-recommends`
- Ajouter votre user au groupe docker, ATTENTION ça lui donne les droits admin
  `sudo usermod -aG docker $USER`
- Installer git
  `sudo apt install git`

Développement local sur Windows avec X11 forwarding

- Vérifier les prérequis à WSL2 de votre Windows
- Installer Docker Desktop
- Vérifier que Docker est configuré et fonctionne avec WSL2
- Installer une distribution Linux via WSL2, par exemple Ubuntu 24 depuis le Microsoft Store
- Lancer un Terminal Ubuntu sur WSL2 (taper `ubuntu` depuis le menu démarrer)

- Installer git
  `sudo apt install git`

Développement sur serveur distant

- Installer une distribution Linux Server, par exemple Yunohost 12
- Installer Docker en mode rootless
- Créer un utilisateur YunoHost, lui donner un accès ssh et les droits appropriés (ne pas l'ajouter au groupe admin YunoHost)
- NE PAS ajouter votre user au groupe docker, car ça lui donnerait des droits admin
  ~~`sudo usermod -aG docker $USER`~~
- Lancer les commandes `docker` avec `sudo`

Avec PWA

> Ici, on suppose qu'on ne fait pas tourner d'algorithmes qui ont besoin de GPU, type Machine Learning, dans le conteneur.

Une première approche est de séparer la partie calcul CPU de la partie graphique GPU. On fait tourner les calculs dans le conteneur. On fait tourner les applications graphiques dans le navigateur Web de l'hôte via une PWA, par exemple via `gzweb` pour gazebo. Il nous faut alors un navigateur qui supporte l'accélération graphique des applications Web, seuls Chrome et dérivés supportent le WebGL :

- Installer Chrome sur Windows ou Chromium sur Linux, de préférence depuis le PPA
- Vérifier que WebGL est activé en vérifiant que le cube tourne sur https://get.webgl.org/

> L'énorme avantage est que l'environnement de développement Docker n'a pas besoin d'accélération graphique. Pas besoin de perdre du temps à essayer de passer la carte graphique au conteneur. Un serveur sans accélération graphique suffit.

https://discourse.ros.org/t/mini-workshop-developing-and-teaching-ros-from-a-web-browser-using-dev-containers-and-pwas/31533

https://github.com/rplayground/sandbox

```
source /opt/ros/$ROS_DISTRO/setup.bash
source /usr/share/gazebo/setup.sh
GAZEBO_MODEL_PATH=$GAZEBO_MODEL_PATH:$(find /opt/ros/$ROS_DISTRO/share \
```

```
  -mindepth 1 -maxdepth 2 -type d -name "models" | paste -s -d: -)
ros2 launch ./launch/security_demo_launch.py \
  use_rviz:=False headless:=True
```

Pour un développement local

- Installer Visual Studio Code, sur Linux depuis Snap ou le PPA Microsoft Noter que Codium sur Linux ne supporte par l'extension Dev Container nécessaire
- Désactiver la collecte des données : `Settings > Telemetry > Telemetry Level > off`
- Installer Docker ~~avec le support de l'accélération graphique~~, cf. ci-dessus
- Installer l'extension `Dev Container`. Noter que les extensions nécessaires au développement de ROS seront installées dans le VSCode-Server du container
- Sur Windows : Lancer un Terminal Ubuntu sur WSL2 (taper ubuntu depuis le menu démarrer)
- Sur Linux : Lancer un Terminal
- Cloner le dépôt du Dev Container
  `git clone https://github.com/rplayground/sandbox`
- Se placer dans le dossier
  `cd ~/sandbox`
- Ouvrir le dossier dans Visual Studio Code
  `code .`
- Lancer `Dev Container: Reopen in Container`

Pour un développement distant sur un serveur/PC

- Installer le pack d'extensions `Remote Development`
- Se connecter à la machine de développement distante depuis Visual Studio Code
  - Ouvrir la Command Palette `Ctrl+Shift+P`
  - Lancer `Remote-SSH: Connect to host...`
  - Configurer la connexion ssh par mot-de-passe ou clé si ce n'a pas encore été fait
- Cloner le dépôt https://github.com/rplayground/sandbox
- Ouvrir le dossier sandbox
- Lancer `Dev Container: Reopen in Container`

Tester l'installation

- Le dépôt https://github.com/rplayground/sandbox?tab=readme-ov-file#demo fournit un environnement préconfiguré avec
  - Les paquets Nav2
  - Les paquets turtlebot3_simulations
  - Le serveur Gazebo et le client web gzweb
  - Des utilitaires pour le démarrage des interfaces graphiques dans le navigateur
- Ouvrir un nouveau Terminal dans VisualStudio Code
- Configurer le Bash et lancer la démo de sécurité :

```
source /opt/ros/$ROS_DISTRO/setup.bash
source /usr/share/gazebo/setup.sh
GAZEBO_MODEL_PATH=$GAZEBO_MODEL_PATH:$(find /opt/ros/$ROS_DISTRO/share \
  -mindepth 1 -maxdepth 2 -type d -name "models" | paste -s -d: -)
ros2 launch ./launch/security_demo_launch.py \
  use_rviz:=False headless:=True
```

- Depuis la palette de commandes `F1` , taper `Tasks: Run Task` et sélectionner `Start Visualizations`
- Depuis le panneau des Ports, cliquer `Open in Browser` pour le port `8080` et ouvrir le lien dans Chrome/Chromium

Il est possible de lancer d'autres environnements de simulation, par ex. turtlebot3_simulations :

- On choisi le modèle de turtlebot3 et on lance gazebo en mode headless

  ```
  export ROS_DOMAIN_ID=30 #TURTLEBOT3
  export LDS_MODEL=LDS-01
  export TURTLEBOT3_MODEL=burger
  ros2 launch turtlebot3_gazebo turtlebot3_world.launch.py \
    use_rviz:=False headless:=True
  ```

Comprendre les spécificités du container sandbox

- On remarque qu'on est connecté en `root` et que la sandbox est installée dans un workspace d'overlay
  `root@iha-portrob-1:/opt/overlay_ws/src/sandbox#`
- Le fichier de configuration du bash `cat /root/.bashrc` n'a pas été modifié pour sourcer les exécutables ros
- La distribution ROS installée est
  `echo $ROS_DISTRO`
  `humble`
- A chaque réinitialisation du container et ouverture de nouveau Terminal il faut donc lancer :

  ```
  source /opt/ros/$ROS_DISTRO/setup.bash
  source /usr/share/gazebo/setup.sh
  GAZEBO_MODEL_PATH=$GAZEBO_MODEL_PATH:$(find /opt/ros/$ROS_DISTRO/share \
    -mindepth 1 -maxdepth 2 -type d -name "models" | paste -s -d: -)
  ```

- Que nous dit `devcontainer.json`

```
{
    "name": "Sandbox",
    "image": "ghcr.io/rplayground/sandbox:main",
    // "build": {
    //     "dockerfile": "../Dockerfile",
    //     "context": "..",
    //     "target": "visualizer",
    //     "cacheFrom": "ghcr.io/rplayground/sandbox:main"
    // },
    "runArgs": [
        // "--cap-add=SYS_PTRACE", // enable debugging, e.g. gdb
        // "--ipc=host", // shared memory transport with host, e.g. rviz GUIs
        // "--network=host", // network access to host interfaces, e.g. eth0
        // "--pid=host", // DDS discovery with host, without --network=host
        // "--privileged", // device access to host peripherals, e.g. USB
        // "--security-opt=seccomp=unconfined", // enable debugging, e.g. gdb
    ],
    "workspaceFolder": "/opt/overlay_ws/src/sandbox",
    "workspaceMount":
"source=${localWorkspaceFolder},target=${containerWorkspaceFolder},type=bind",
    "onCreateCommand": ".devcontainer/on-create-command.sh",
    "updateContentCommand": ".devcontainer/update-content-command.sh",
    "postCreateCommand": ".devcontainer/post-create-command.sh",
    "customizations": {
        "codespaces": {
            "openFiles": [
                "README.md"
            ]
        },
        "vscode": {
            "settings": {},
            "extensions": [
                "eamodio.gitlens",
                "GitHub.copilot",
                "ms-iot.vscode-ros",
                "streetsidesoftware.code-spell-checker"
            ]
        }
    }
}
```

- Que nous dit `Dockerfile`

Installation de Docker avec accélération graphique

> Ici, on peut faire tourner dans le conteneur des algorithmes qui ont besoin de GPU, type Machine Learning.

AMD/Intel sous Linux

NVidia Sous Linux

- Installer le Driver propriétaire NVidia (si les driver proprio n'ont pas été autorisés à l'installation d'Ubuntu)
- `sudo apt install curl`
- Installer le Nvidia Container Toolkit

```
curl -fsSL https://nvidia.github.io/libnvidia-container/gpgkey | sudo gpg --dearmor -o /usr/share/keyrings/nvidia-container-toolkit-keyring.gpg \
  && curl -s -L https://nvidia.github.io/libnvidia-container/stable/deb/nvidia-container-toolkit.list | \
    sed 's#deb https://#deb [signed-by=/usr/share/keyrings/nvidia-container-toolkit-keyring.gpg] https://#g' | \
    sudo tee /etc/apt/sources.list.d/nvidia-container-toolkit.list && \
    sudo apt-get update && sudo apt-get install nvidia-container-toolkit
```

- Redémarrer le PC
- Lancer docker avec l'argument `--gpus=all` pour tester qu'il a bien accès au GPU `docker run --rm -it --gpus=all nvcr.io/nvidia/k8s/cuda-sample:nbody nbody -gpu -benchmark`
- Le résultat suivant indique que la carte graphique dédiée `Nvidia Quadro P620` est bien exploitée pour les calculs :

```
> Windowed mode
> Simulation data stored in video memory
> Single precision floating point simulation
```

```
> 1 Devices used for simulation
GPU Device 0: "Pascal" with compute capability 6.1

> Compute 6.1 CUDA device: [Quadro P620]
4096 bodies, total time for 10 iterations: 4.417 ms
= 37.987 billion interactions per second
= 759.750 single-precision GFLOP/s at 20 flops per interaction
```

- Si ça ne s'affiche pas : https://docs.nvidia.com/datacenter/cloud-native/container-toolkit/latest/install-guide.html#configuring-docker

NVidia Sous Linux Public Server (rootless docker)

https://docs.nvidia.com/datacenter/cloud-native/container-toolkit/latest/install-guide.html#rootless-mode

NVidia Sous Windows

https://innovation.iha.unistra.fr/books/robotique-open-source/page/installation-pc-ros2#bkmrk-acc%C3%A9l%C3%A9ration-gpu-pou

Lancement Dev Container avec accélération graphique

- Ajouter dans `.devcontainer/devcontainer.json`

```
{
    "name": "Sandbox",
    "hostRequirements": {
        "gpu": "optional" // switch on CPU if no (NVidia?) GPU available
    },
    "runArgs": [
        "--gpus=all" // enable NVidia GPU with NVidia driver
        // "--device=/dev/dri" // enable Intel/AMD GPU
}
```

Avec X11 Forwarding

On peut aussi afficher les fenêtres graphiques des applications qui tournent dans le Container directement sur l'hôte, par ex. `gazebo-client`

Pour cela il faut :

- Installer dans le conteneur le gestionnaire de fenêtres x11-apps (partie serveur)
- Avoir accès à un gestionnaire de fenêtres X11 sur l'hôte (partie client)
- Configurer correctement l'hôte et le Dev Container pour que le conteneur ait accès au client X11
- Cela peut poser des problèmes de sécurité car le conteneur risque d'avoir accès à tout ce qu'il se passe sur l'écran de l'hôte

> On ne peut plus se contenter d'une distribution Linux Serveur type YunoHost pour le développement distant

Les calculs graphiques se font alors dans le container ??

Avec VNC

Ressources

https://articulatedrobotics.xyz/tutorials/docker/dev-containers/

Nav2, container, PWA https://discourse.ros.org/t/repeatable-reproducible-and-now-accessible-ros-development-via-dev-containers/31398 https://github.com/ros-navigation/docs.nav2.org/blob/master/development_guides/devcontainer_docs/devcontainer_guide.md

---

Revision #9
Created 4 September 2025 15:02:20 by admin_idf
Updated 10 September 2025 10:21:32 by admin_idf