MESURE DES PERFORMANCES

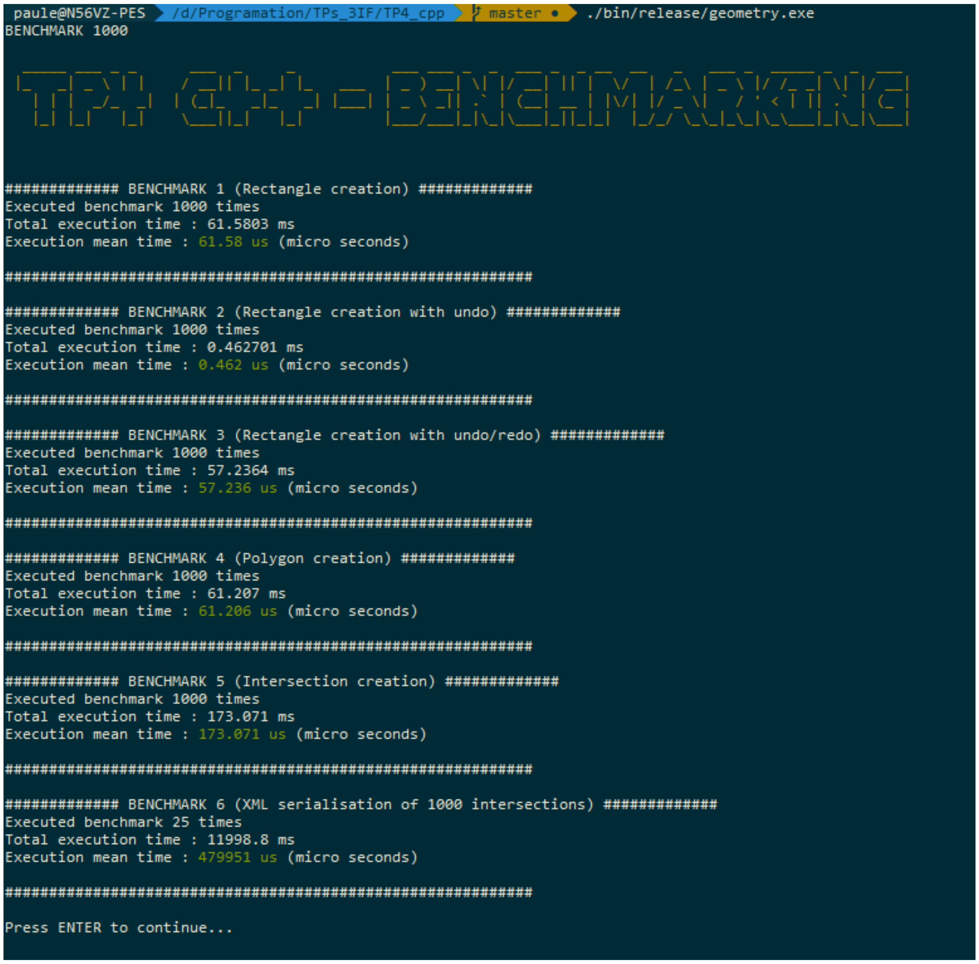
La mesure des performances de l'application se fait grâce à des mesures de temps dans 'Benchmark.h'. L'utilisateur peut utiliser la commande 'BENCHMARK' suivie du nombre de répétitions des fonctions décrivant les tests de performances.

Les tests de performance sont définis dans des fonctions lambda données en paramètre à la fonction '`benchmark`' qui mesure le temps d'exécution du test et donne le temps pour une seule opération (création d'un rectangle, polygone, sauvegarde, …). Voici les tests de performance exécutés lors de l'appel de la commande 'BENCHMARK' (le nombre des répétitions est dans la valeur '`repetition_count`') :

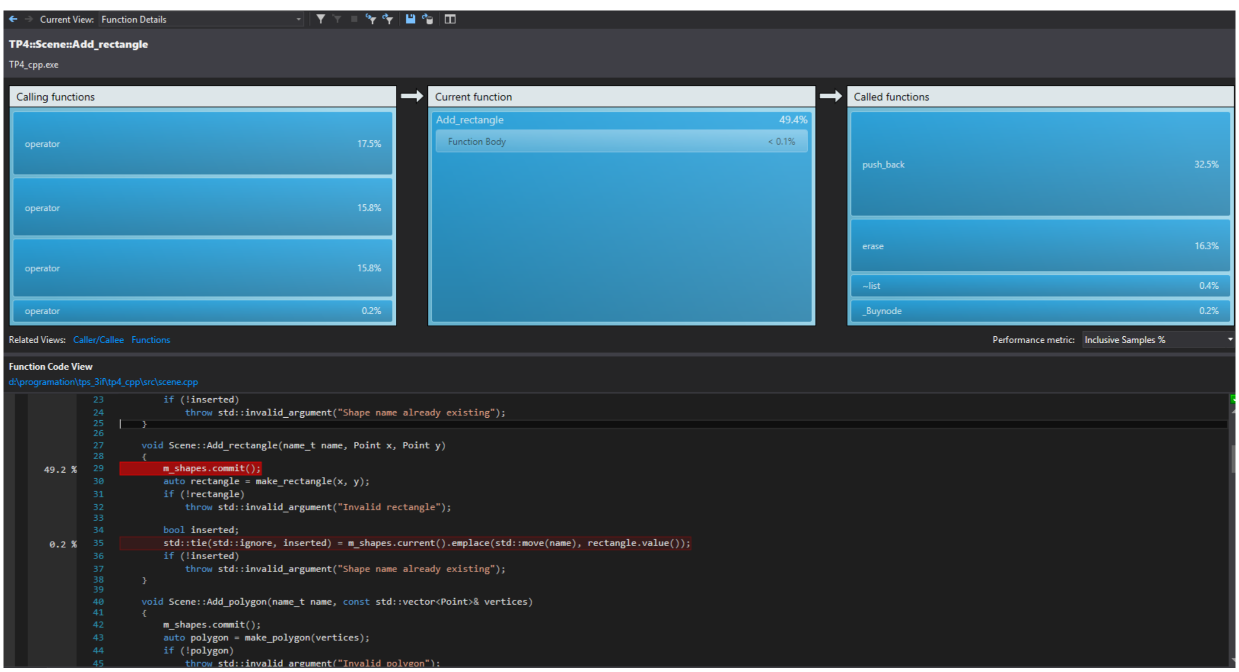
```
// Creation of 'repetition_count' rectangles
benchmark("Rectangle creation", repetition_count, [&geometry_scene, repetition_count]()
{
	static const TP4::Point p1{ 1, 8 };
	static const TP4::Point p2{ 4, 2 };

	unsigned int i = repetition_count + 1;
	while (--i)
		geometry_scene.Add_rectangle("rect" + std::to_string(i), p1, p2);
});

geometry_scene.ClearAll();
// ... (other tests)

// Creation of 'repetition_count' intersections of rectangles and segments
benchmark("Intersection creation", repetition_count, [&geometry_scene, repetition_count]()
{
	static const TP4::Point p1{ 1, 8 };
	static const TP4::Point p2{ 4, 2 };

	unsigned int i = repetition_count + 1;
	while (--i)
	{
		auto i_str = std::to_string(i);
		geometry_scene.Add_segment("seg" + i_str, p1, p2);
		geometry_scene.Add_rectangle("rect" + i_str, p1, p2);
		geometry_scene.Intersect("inter" + i_str, { "seg" + i_str, "rect" + i_str });
	}
});

// Save and load 'repetition_count/40' XML files
repetition_count /= 40;
benchmark("XML serialisation of " + std::to_string(40*repetition_count) + " intersections",
	repetition_count, [&geometry_scene, repetition_count]()
{
	unsigned int i = repetition_count + 1;
	while (--i)
	{
		geometry_scene.Save("test.xml");
		geometry_scene.ClearCurrentState();
		geometry_scene.Load("test.xml");
	}
});
```

Le dernier test sauvegarde et charge un fichier XML contenant les formes du test précédent ('`repetition_count`' intersections).

L’appel à la commande ‘BENCHMARK 1000’ crée la sortie suivante :



On se rend compte rapidement que le temps d’exécution d’une opération individuelle (temps moyen) augmente si l’on fait plus de répétitions. Cela s’explique par le fait que dans ces tests, les formes sont très simples et donc la majorité du temps de calcul est dédié à l’historique qui doit copier tous les shared_ptr<History_shape> (copie en surface). On peut mieux s’en rendre compte grace aux outils de profilage de visual studio. Voici une capture d’écran montrant l’utilisation du CPU de chaque méthodes lors de l’execution du benhmark (sans la (de)serialisation des fichiers de sauvegarde) :



On y voit clairement que la majorité du temps est passé dans la fonction commit (où la majorité du temps de calcul est utilisé par la copie des shared_ptr).

Conclusion

Ces benchmarks nous permettent de voir facilement quelles sont les opérations couteuses et de valider la conception du projet. C'est également l'occasion de s'habituer à l'utilisation d'outils de profilage et de prendre conscience de l'impact de nos choix de conception.

En réalité, si ce programme était dédié à une application graphique classique, on ne vas pas créer à répétition des formes géométriques mais on veut que la sauvegarde (serialisation) et l'anulation d'actions (Undo) soit rapide. Or, la sauvegarde reposant sur boost::serialization, la serialisation peut-être multithread (dépend de la version de la librairie boost utilisée) et les performances du benchmark sont satisfaisantes. Enfin, l'anulation et la restitution d'action est également rapide car il s'agit juste d'incrémenter ou décrémenter un itérateur sur un vecteur d'états de l'historique.