مقدمة في محاكاة Chip-8 باستخدام لغة البرمجة Rust

by aquova

February 2023 15

مقدمة إلى محاكاة ألعاب الفيديو باستخدام Rust

تطوير محاكي لألعاب الفيديو أصبح مشروعًا هواية شائعًا بشكل متزايد بين المطورين. يتطلب هذا المشروع معرفة بالأجهزة منخفضة المستوى، ولغات البرمجة الحديثة، وأنظمة الرسومات لإنشائه بنجاح. يعتبر هذا المشروع تعليميًا ممتازًا؛ ليس فقط لأنه يحتوي على أهداف واضحة، ولكنه أيضًا مجزٍ للغاية عندما تتمكن من تشغيل الألعاب على محاكي قمت بكتابته بنفسك. أنا ما زلت مطور محاكاة جديد نسبيًا، لكنني لم أكن لأصل إلى ما أنا عليه الآن لولا الأدلة والبرامج التعليمية الرائعة المتاحة على الإنترنت. لذلك، أردت أن أرد الجميل للمجتمع من خلال كتابة دليل يحتوي على بعض الحيل التي تعلمتها، على أمل أن يكون مفيدًا لشخص آخر.

مقدمة عن Chip-8

نظامنا المستهدف هو Chip-8. أصبح Chip-8 بمثابة "Hello World" لتطوير المحاكاة. بينما قد تكون مغريًا للبدء بشيء أكثر إثارة مثل NES أو Game Boy، إلا أن هذه الأنظمة أكثر تعقيدًا من Chip-8. يحتوي Chip-8 على شاشة أحادية اللون بدقة 1 بت، وصوت بسيط أحادي القناة، و35 تعليمة فقط (مقارنة بحوالي 500 تعليمة في Game Boy)، ولكن سنتحدث عن ذلك لاحقًا. سيغطي هذا الدليل التفاصيل التقنية لـ Chip-8، وما هي أنظمة الأجهزة التي تحتاج إلى محاكاتها

وكيفية ذلك، وكيفية التفاعل مع المستخدم. سيركز هذا الدليل على مواصفات Chip-8 الأصلية، ولن يتم تنفيذ أي من الامتدادات العديدة التي تم اقتراحها، مثل Super Chip-8 أو Chip-16 أو XO-Chip؛ حيث تم إنشاؤها بشكل مستقل عن بعضها البعض، وبالتالي تضيف ميزات متناقضة.

المواصفات التقنية لـ Chip-8

- شاشة أحادية اللون بدقة 64x32، يتم الرسم عليها عبر أشكال (sprites) بعرض 8 بكسل وارتفاع يتراوح بين 1 و16 بكسل.
- ستة عشر سجلًا عامًا بعرض 8 بت، يُشار إليها بـ V0 حتى VF. يعمل VF أيضًا كسجل علم (flag register) لعمليات الفائض (overflow).
- عداد برنامج (program counter) بعرض 16 بت.
- سجل واحد بعرض 16 بت يُستخدم كمؤشر للوصول إلى الذاكرة، يُسمى *سجل I*.
- كمية غير موحدة من الذاكرة العشوائية (RAM)، ولكن معظم المحاكيات تخصص 4 كيلوبايت.
- مكدس (stack) بعرض 16 بت يُستخدم لاستدعاء العودة من الإجراءات الفرعية (subroutines).

- إدخال لوحة مفاتيح مكونة من 16 مفتاحًا.
- سجلين خاصين ينخفضان كل إطار ويتم تشغيلهما عند الوصول إلى الصفر:
  - مؤخر الوقت (Delay timer): يُستخدم للأحداث الزمنية في اللعبة.
  - مؤخر الصوت (Sound timer): يُستخدم لتشغيل صوت التنبيه.

مقدمة عن Rust

يمكن كتابة المحاكيات بلغات برمجة عديدة. يستخدم هذا الدليل لغة البرمجة Rust، على الرغم من أن الخطوات الموضحة هنا يمكن تطبيقها بأي لغة. تقدم Rust العديد من المزايا الرائعة؛ فهي لغة مكتوبة (compiled) تدعم منصات رئيسية ولديها مجتمع نشط من المكتبات الخارجية التي يمكن استخدامها في مشروعنا. تدعم Rust أيضًا التجميع لـ WebAssembly، مما يسمح لنا بإعادة تجميع الكود للعمل في المتصفح مع تعديلات بسيطة. يفترض هذا الدليل أنك تفهم أساسيات لغة Rust والبرمجة بشكل عام. سأشرح الكود أثناء تقدمنا، ولكن نظرًا لأن Rust تتمتع بمنحنى تعليمي مرتفع، أوصي بقراءة والرجوع إلى الكتاب الرسمي لـ Rust لأي مفاهيم غير مألوفة أثناء تقدم الدليل. يفترض هذا الدليل أيضًا أنك قمت بتثبيت Rust وأنها تعمل بشكل صحيح. يرجى الرجوع إلى تعليمات التثبيت لمنصتك إذا لزم الأمر.

ما ستحتاج إليه

قبل أن تبدأ، يرجى التأكد من أن لديك أو قمت بتثبيت العناصر التالية.

محرر نصوص

يمكن استخدام أي محرر نصوص للمشروع، ولكن هناك محرران أوصي بهما حيث يقدمان ميزات لـ Rust مثل تمييز الصيغة (syntax highlighting)، اقتراحات الكود، ودعم المصحح (debugger).

- Visual Studio Code هو المحرر الذي أفضله لـ Rust، بالاشتراك مع إضافة rust-analyzer.
- بينما لا تقدم JetBrains بيئة تطوير متكاملة (IDE) مخصصة لـ Rust، إلا أن هناك إضافة لـ Rust للعديد من منتجاتها الأخرى. توفر الإضافة لـ CLion ميزات إضافية مثل دعم المصحح المدمج. ضع في اعتبارك أن CLion منتج مدفوع، على الرغم من أنه يقدم نسخة تجريبية لمدة 30 يومًا وفترات مجانية ممتدة للطلاب.

إذا كنت لا تفضل أيًا من هذه الخيارات، تتوفر إضافات للصيغة والاكتمال التلقائي لـ Rust للعديد من المحررات الأخرى، ويمكن تصحيح الأخطاء بسهولة باستخدام العديد من المصححات الأخرى مثل gdb.

ROMs للاختبار

المحاكي ليس مفيدًا إذا لم يكن لديك شيء لتشغيله! تم تضمين العديد من برامج Chip-8 الشائعة مع الكود المصدري لهذا الكتاب، ويمكن أيضًا العثور عليها

هنا. سيتم عرض بعض هذه الألعاب كأمثلة خلال هذا الدليل.

أشياء أخرى

عناصر أخرى قد تكون مفيدة أثناء تقدمنا:

- يرجى تحديث معرفتك بـ النظام الست عشري (hexadecimal) إذا كنت لا تشعر بالراحة مع هذا المفهوم. سيتم استخدامه بشكل مكثف خلال هذا المشروع.
- ألعاب Chip-8 تكون بتنسيق ثنائي (binary)، وغالبًا ما يكون من المفيد أن تكون قادرًا على عرض القيم الست عشرية الخام أثناء تصحيح الأخطاء. عادةً لا تدعم محررات النصوص القياسية عرض الملفات بالنظام الست عشري، بل يتطلب ذلك محرر ست عشري متخصص hex editor. العديد منها يقدم ميزات مشابهة، لكني أفضل شخصيًا Reverse Engineer’s Hex Editor.

لنبدأ!

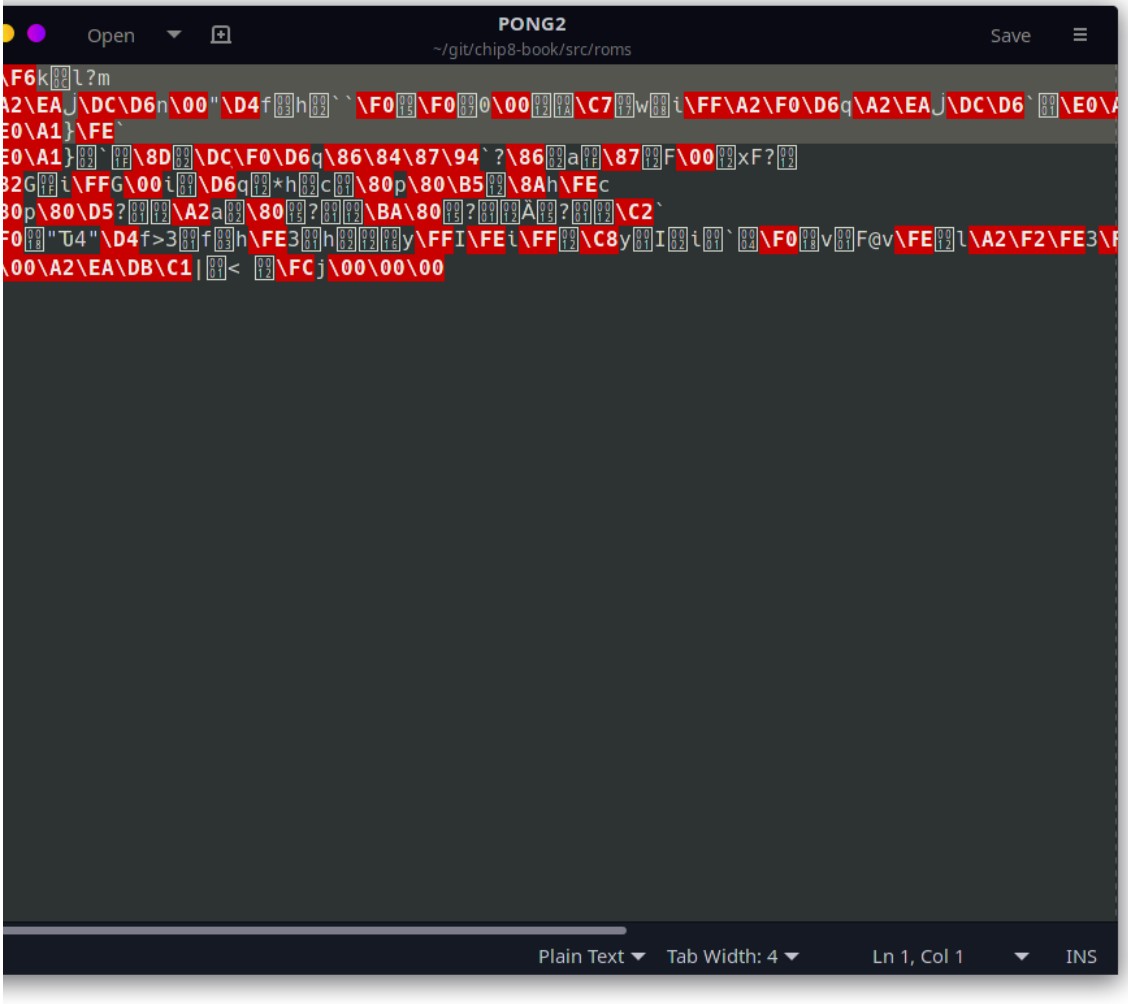
أساسيات المحاكاة

هذا الفصل الأول هو نظرة عامة على مفاهيم تطوير المحاكاة ونظام Chip-8. الفصول اللاحقة ستتضمن تنفيذًا للكود بلغة Rust. إذا لم تكن مهتمًا بـ Rust أو تفضل العمل دون أمثلة، فإن هذا الفصل سيعطي مقدمة عن الخطوات التي يقوم بها النظام الحقيقي، وما الذي نحتاج إلى محاكاته بأنفسنا، وكيف تتلاءم جميع الأجزاء معًا. إذا كنت جديدًا تمامًا على هذا الموضوع، فإن هذا الفصل سيوفر المعلومات التي تحتاجها لفهم ما ستقوم بتنفيذه.

ما الذي يوجد في لعبة Chip-8؟

لنبدأ بسؤال بسيط. إذا أعطيتك ملف ROM[1] لـ Chip-8، ما الذي يحتويه بالضبط؟ يحتاج الملف إلى أن يحتوي على كل منطق اللعبة والأصول الرسومية اللازمة لجعل اللعبة تعمل على الشاشة، ولكن كيف يتم تنظيم كل ذلك؟

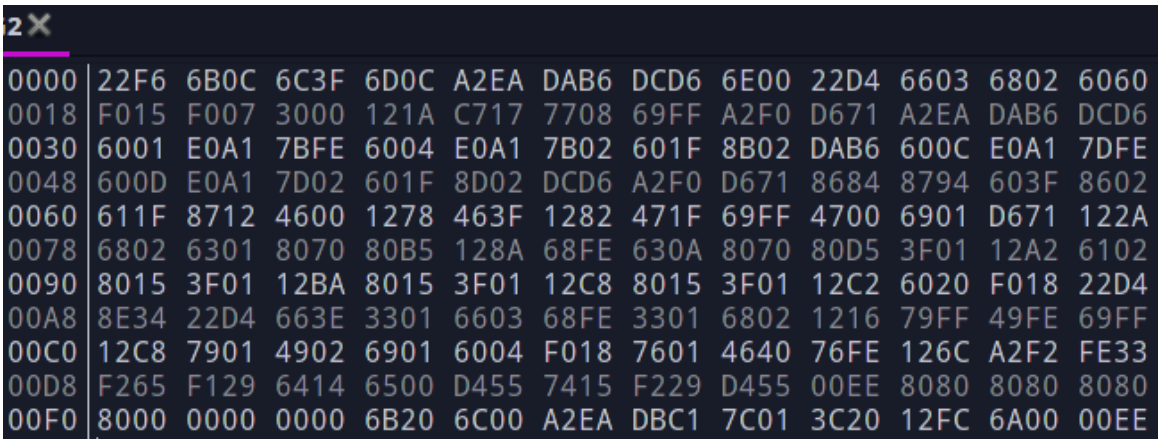
هذا سؤال أساسي؛ هدفنا هو قراءة هذا الملف وجعل برنامجه يعمل. حسنًا، هناك طريقة واحدة لمعرفة ذلك، لذا دعنا نحاول فتح لعبة Chip-8 في محرر نصوص (مثل Notepad أو TextEdit أو أي محرر آخر تفضله). في هذا المثال، أستخدم لعبة `roms/PONG2` المضمنة مع هذا الدليل. ما لم تكن تستخدم محررًا متطورًا جدًا، فمن المحتمل أن ترى شيئًا مشابهًا للشكل 1.



ملف ROM لـ Chip-8 بشكل خام

هذا لا يبدو مفيدًا جدًا. في الواقع، يبدو تالفًا. لا تقلق، لعبتك (على الأرجح) بخير. جميع برامج الكمبيوتر هي في جوهرها مجرد أرقام محفوظة في ملف. يمكن أن تعني هذه الأرقام أي شيء تقريبًا؛ فقط مع السياق يمكن للكمبيوتر أن يعطي معنى لهذه القيم. تم تصميم محرر النصوص الخاص بك لعرض وتحرير النصوص بلغة مثل الإنجليزية، وبالتالي سيحاول تلقائيًا استخدام سياق يركز على اللغة مثل ASCII.

لقد تعلمنا أن ملف Chip-8 الخاص بنا ليس مكتوبًا باللغة الإنجليزية العادية، وهو ما كنا نستطيع افتراضه بالفعل. إذن كيف يبدو؟ لحسن الحظ، توجد برامج لعرض المحتويات الخام للملف، لذا سنحتاج إلى استخدام أحد "محررات الست عشري" لعرض المحتويات الفعلية لملفنا.



0000	22F6	6B0C	6C3F	6D0C	A2EA	DAB6	DCD6	6E00	22D4	6603	6802	6060
0018	F015	F007	3000	121A	C717	7708	69FF	A2F0	D671	A2EA	DAB6	DCD6
0030	6001	E0A1	7BFE	6004	E0A1	7B02	601F	8B02	DAB6	600C	E0A1	7DFE
0048	600D	E0A1	7D02	601F	8D02	DCD6	A2F0	D671	8684	8794	603F	8602
0060	611F	8712	4600	1278	463F	1282	471F	69FF	4700	6901	D671	122A
0078	6802	6301	8070	80B5	128A	68FE	630A	8070	80D5	3F01	12A2	6102
0090	8015	3F01	12BA	8015	3F01	12C8	8015	3F01	12C2	6020	F018	22D4
00A8	8E34	22D4	663E	3301	6603	68FE	3301	6802	1216	79FF	49FE	69FF
00C0	12C8	7901	4902	6901	6004	F018	7601	4640	76FE	126C	A2F2	FE33
00D8	F265	F129	6414	6500	D455	7415	F229	D455	00EE	8080	8080	8080
00F0	8000	0000	0000	6B20	6C00	A2EA	DBC1	7C01	3C20	12FC	6A00	00EE

ملف ROM لـ Chip-8

في الشكل 2، الأرقام على يسار الخط العمودي هي قيم *الإزاحة* (offset)، وهي عدد البايتات التي نبتعدها عن بداية الملف. على الجانب الأيمن توجد القيم الفعلية المخزنة في الملف. يتم عرض كل من الإزاحات وقيم البيانات بالنظام الست عشري (سنتعامل مع النظام الست عشري بشكل كبير).

حسنًا، لدينا أرقام الآن، وهذا تحسن. إذا كانت هذه الأرقام لا تتوافق مع الحروف الإنجليزية، فماذا تعني؟ هذه هي *التعليمات* لوحدة المعالجة المركزية (CPU) لـ Chip-8. في الواقع، كل تعليمة تتكون من بايتين، ولهذا قمت بتجميعها في أزواج في لقطة الشاشة.

ما هي وحدة المعالجة المركزية (CPU)؟

دعني أخذ لحظة لوصف الوظيفة التي توفرها وحدة المعالجة المركزية (CPU) لأولئك الذين ليسوا على دراية بها. وحدة المعالجة المركزية، لأغراضنا، تقوم بالعمليات الحسابية. هذا كل شيء. عندما أقول "تقوم بالعمليات الحسابية"، فإن هذا يشمل العناصر المعتادة مثل الجمع والطرح والتحقق مما إذا كان الرقمان متساويين أم لا. هناك أيضًا عمليات إضافية مطلوبة لتشغيل اللعبة، مثل القفز إلى أقسام مختلفة من الكود، أو جلب الأرقام وحفظها. ملف اللعبة يتكون بالكامل من عمليات حسابية يجب على وحدة المعالجة المركزية تنفيذها.

جميع العمليات الحسابية التي يمكن لـ Chip-8 تنفيذها لها رقم مقابل، يسمى *رمز العملية* (opcode). يمكن رؤية قائمة كاملة برموز عمليات Chip-8 على هذه الصفحة. عندما يحين وقت تنفيذ تعليمة أخرى (يشار إليها أيضًا باسم *tick* أو *cycle*)، ستقوم المحاكاة بجلب رمز العملية التالي من ملف ROM الخاص باللعبة وتنفيذ العملية المحددة في جدول رموز العمليات؛ سواء كانت جمعًا أو طرحًا أو تحديثًا لما يتم رسمه على الشاشة، أو أي شيء آخر.

ماذا عن المعاملات؟ ليس كافيًا أن نقول "حان وقت الجمع"، بل تحتاج إلى رقمين لتجمعهما معًا، بالإضافة إلى مكان لوضع الناتج عند الانتهاء. تقوم الأنظمة الأخرى بذلك بشكل مختلف، ولكن بايتان لكل عملية تعطي الكثير من الأرقام المحتملة، أكثر بكثير من 35 عملية يمكن لـ Chip-8 تنفيذها بالفعل. يتم استخدام الأرقام الإضافية لتضمين معلومات إضافية في رمز العملية. يختلف التخطيط

الدقيق لهذه المعلومات بين رموز العمليات. يستخدم جدول رموز العمليات `N` للإشارة إلى الأرقام الست عشرية الحرفية. `N` لرقم واحد، `NN` لرقمين، و`NNN` لثلاثة أرقام حرفية، أو لتحديد *سجل* عبر `X` أو `Y`.

ما هي السجلات (Registers)؟

وهذا يقودنا إلى موضوعنا التالي. ما هي السجلات؟ *السجل* هو مكان مخصص لتخزين بايت واحد لاستخدامه في التعليمات. قد يبدو هذا مشابهًا للذاكرة العشوائية (RAM)، وبعض النواحي هو كذلك. بينما تعد الذاكرة العشوائية مكانًا كبيرًا لتخزين البيانات، فإن السجلات عادةً ما تكون قليلة العدد، ولها أسماء محددة، ويتم استخدامها مباشرة عند تنفيذ رمز العملية. بالنسبة للعديد من أجهزة الكمبيوتر، إذا كنت تريد استخدام قيمة في الذاكرة العشوائية، فيجب نسخها إلى سجل أولاً.

يحتوي Chip-8 على ستة عشر سجلًا يمكن للمبرمج استخدامها بحرية، تُسمى من V0 إلى VF (0-15 بالنظام الست عشري). كمثال على كيفية عمل السجلات، دعنا ننظر إلى إحدى عمليات الجمع في جدول رموز العمليات. هناك عملية لجمع قيم سجلين معًا، VX += VY، مشفرة كـ `8XY4`. يتم استخدام `8XY4` لمطابقة رموز العمليات. إذا بدأ رمز العملية الحالي بـ 8 وانتهى بـ 4، فإن هذه هي العملية المطابقة. الرقمان في المنتصف يحددان السجلات التي سنستخدمها. لنفترض أن رمز العملية لدينا هو `8124`. يبدأ بـ 8 وينتهي بـ 4، لذا نحن في المكان الصحيح. بالنسبة لهذه التعليمات، سنستخدم القيم المخزنة في V1 وV2، حيث تتطابق

مع الرقمين الآخرين. لنفترض أن V1 يحتوي على 5 وV2 يحتوي على 10، فإن هذه العملية ستجمع هذين الرقمين وتستبدل ما كان في V1، وبالتالي سيحتوي V1 الآن على 15.

يحتوي Chip-8 على عدد قليل من السجلات الإضافية، ولكنها تخدم أغراضًا محددة للغاية. أحد أهمها هو *عداد البرنامج* (PC)، والذي يمكنه تخزين قيمة بعرض 16 بت. لقد أشرت بشكل غامض إلى "رمز العملية الحالي"، ولكن كيف نتابع أين نحن؟ المحاكاة الخاصة بنا تحاكي جهاز كمبيوتر يعمل على برنامج. تحتاج إلى البدء من بداية اللعبة والانتقال من رمز عملية إلى آخر، وتنفيذ التعليمات كما يتم إخبارها. يحتفظ PC بفهرس التعليمات التي نعمل عليها حاليًا. لذا سيبدأ من البايت الأول من اللعبة، ثم ينتقل إلى الثالث (تذكر أن جميع رموز العمليات تتكون من بايتين)، وهكذا دواليك. يمكن لبعض التعليمات أيضًا أن تخبر PC بالانتقال إلى مكان آخر في اللعبة، ولكن بشكل افتراضي سيتحرك للأمام رمزًا تلو الآخر.

ما هي الذاكرة العشوائية (RAM)؟

لدينا ستة عشر سجلًا V، ولكن حتى لنظام بسيط مثل Chip-8، نرغب حقًا في القدرة على تخزين أكثر من 16 رقمًا في وقت واحد. هنا تأتي الذاكرة العشوائية (RAM). ربما تكون على دراية بها في سياق جهاز الكمبيوتر الخاص بك، ولكن الذاكرة العشوائية هي مصفوفة كبيرة من الأرقام يمكن لوحدة المعالجة المركزية استخدامها كما تشاء. Chip-8 ليس نظامًا ماديًا، لذا لا يوجد كمية قياسية من

الذاكرة العشوائية التي يجب أن يحتويها. ومع ذلك، اتفق مطورو المحاكاة على 4096 بايت (4 كيلوبايت)، لذا سيكون لنظامنا القدرة على تخزين ما يصل إلى 4096 رقمًا بعرض 8 بت في ذاكرته العشوائية، وهو أكثر بكثير مما ستستخدمه معظم الألعاب.

الآن، حان وقت تفصيل مهم: وحدة المعالجة المركزية لـ Chip-8 لديها وصول حر للقراءة والكتابة في الذاكرة العشوائية كما تشاء. ومع ذلك، لا يمكنها الوصول مباشرة إلى ملف ROM الخاص باللعبة. مع اسم مثل "ذاكرة القراءة فقط"، من الآمن افتراض أننا لن نكون قادرين على الكتابة فوقها، ولكن وحدة المعالجة المركزية لا يمكنها القراءة من ROM أيضًا؟ بينما تحتاج وحدة المعالجة المركزية إلى القدرة على قراءة ملف ROM الخاص باللعبة، فإنها تحقق ذلك بشكل غير مباشر، عن طريق نسخ اللعبة بالكامل إلى الذاكرة العشوائية2 عند بدء تشغيل اللعبة. من البطيء وغير الفعال فتح ملف اللعبة فقط لقراءة جزء صغير من البيانات مرارًا وتكرارًا. بدلاً من ذلك، نريد أن نتمكن من نسخ أكبر قدر ممكن من البيانات في الذاكرة العشوائية لاستخدامها بسرعة أكبر. التفصيل الآخر هو أنه بشكل مربك بعض الشيء، لا يتم تحميل بيانات ROM في بداية الذاكرة العشوائية. بدلاً من ذلك، يتم إزاحتها بمقدار 512 بايت (0x200). هذا يعني أن البايت الأول من اللعبة يتم تحميله في بداية الذاكرة العشوائية عند العنوان 0x200. البايت الثاني من ROM يتم تحميله في 0x201، وهكذا.

لماذا لا يقوم Chip-8 بتخزين اللعبة في بداية الذاكرة العشوائية وينتهي الأمر؟ عندما تم تصميم Chip-8، كانت أجهزة الكمبيوتر تحتوي على ذاكرة عشوائية محدودة للغاية، وكان يمكن تشغيل برنامج واحد فقط في وقت معين. تم تخصيص أول 0x200 بايت لتشغيل برنامج Chip-8 نفسه. وبالتالي، يحتاج مطورو محاكاة Chip-8 الحديثة إلى أخذ ذلك في الاعتبار، حيث لا تزال الألعاب تُطور مع هذا المفهوم. سيقوم نظامنا في الواقع باستخدام جزء صغير من هذه المساحة الفارغة، ولكننا سنغطي ذلك لاحقًا.

---

1. ‘ROM’ تعني “ذاكرة القراءة فقط” (Read-only memory). وهي نوع من الذاكرة يتم كتابتها بشكل دائم أثناء التصنيع ولا يمكن تعديلها بواسطة نظام الكمبيوتر. لأغراض هذا الدليل، سيتم استخدام “ملف ROM” بالتبادل مع “ملف اللعبة”.

2. لذلك، فإن ألعاب Chip-8 لها حجم أقصى يبلغ 4 كيلوبايت، وإذا كانت أكبر من ذلك، فلا يمكن تحميلها بالكامل.

الإعداد

نظرًا لأن الهدف النهائي لهذا المشروع هو الحصول على مُحاكي يمكن بناؤه لكل من سطح المكتب ومتصفح الويب، سنقوم ببناء هذا المشروع بشكل غير معتاد قليلاً. بين الإصدارين، يجب أن يكون المحاكي الفعلي لنظام Chip-8 متماثلًا تمامًا، فقط أشياء مثل قراءة الملف وعرض الشاشة ستكون مختلفة بين سطح المكتب والمتصفح. لهذا الغرض، سنقوم بإنشاء الواجهة الخلفية، والتي سأسميها *النواة*، كوحدة مستقلة يمكن استخدامها من قبل واجهاتنا المستقبلية.

انتقل إلى المجلد حيث ستخزن مشروعك وقم بتنفيذ الأمر التالي. لا تقم بتضمين رمز `$`، فهو ببساطة للإشارة إلى أن هذا أمر في الطرفية.

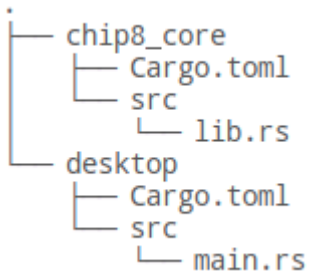
```
$ cargo init chip8_core --lib
```

علم `--lib` يخبر `cargo` بإنشاء مكتبة بدلاً من وحدة تنفيذية. هذا هو المكان الذي سيتم فيه وضع واجهة المحاكي الخلفية. لقد أسميتها `chip8_core` لأغراض هذا المشروع، ولكنك حر في تسميتها كما تشاء.

أما بالنسبة للواجهة الأمامية، سنقوم بإنشاء وحدة إضافية لحفظ هذا الكود:

```
$ cargo init desktop
```

على عكس النواة، فإن هذا ينشئ مشروعًا تنفيذيًا فعليًا. إذا قمت بذلك بشكل صحيح، يجب أن يبدو هيكل المجلدات الخاص بك الآن كما يلي:



هيكل الملفات الأولي

كل ما تبقى هو إخبار واجهة `desktop` الأمامية عن مكان العثور على `chip8_core`.

افتح ملف `desktop/Cargo.toml` وأضف السطر التالي تحت `[dependencies]`:

```
chip8_core = { path = "../chip8_core" }
```

نظرًا لأن `chip8_core` فارغة حاليًا، فإن هذا لا يضيف أي شيء فعليًا، ولكنه شيء سيحتاج إلى القيام به في النهاية. حاول تجميع وتشغيل المشروع، فقط للتأكد من أن كل شيء يعمل. من داخل مجلد `desktop`، قم بتنفيذ:

```
$ cargo run
```

إذا تم الإعداد بشكل صحيح، يجب أن يظهر "Hello, world!". رائع! بعد ذلك، سنبدأ في محاكاة وحدة المعالجة المركزية وبدء إنشاء شيء أكثر إثارة للاهتمام.

تعريف المحاكي

الجزء الأساسي من النظام هو وحدة المعالجة المركزية، لذا سنبدأ من هناك عند إنشاء المحاكي. في الوقت الحالي، سنقوم بتطوير واجهة `chip8_core` الخلفية بشكل أساسي، ثم نعود لتوفير واجهتنا الأمامية عندما نكون قد تقدمنا بشكل

جيد. سنبدأ بالعمل في ملف `chip8_core/src/lib.rs` (يمكنك حذف كود الاختبار الذي تم إنشاؤه تلقائيًا). قبل أن نضيف أي وظائف، دعنا نسترجع بعض المفاهيم حول ما نحن على وشك القيام به.

المحاكاة هي ببساطة تنفيذ برنامج تم كتابته أصلاً لنظام مختلف، لذا فهي تعمل بشكل مشابه جدًا لتنفيذ برنامج كمبيوتر حديث. عند تشغيل أي برنامج قديم، يتم قراءة سطر من الكود، ويتم فهمه من قبل الكمبيوتر لأداء مهمة ما مثل تعديل متغير، أو إجراء مقارنة، أو القفز إلى سطر مختلف من الكود؛ ثم يتم تنفيذ هذا الإجراء، وينتقل التنفيذ إلى السطر التالي لتكرار هذه العملية. إذا كنت قد درست المترجمات، فستعرف أنه عند تشغيل النظام للكود، فإنه لا يقوم بذلك سطرًا بسطر، بل يحول الكود إلى تعليمات يفهمها المعالج، ثم يقوم بتنفيذ هذه الحلقة على تلك التعليمات. هذا بالضبط كيف سيعمل المحاكي الخاص بنا. سنقوم باجتياز القيم واحدة تلو الأخرى في برنامج اللعبة، **جلب** التعليمات المخزنة هناك، **فك تشفير** العملية التي يجب القيام بها، ثم **تنفيذها**، قبل الانتقال إلى التالي. هذه الحلقة *الجلب-فك التشفير-التنفيذ* ستشكل جوهر محاكاة وحدة المعالجة المركزية.

مع أخذ ذلك في الاعتبار، لنبدأ بتعريف فئة ستقوم بإدارة المحاكي الخاص بنا. في `lib.rs`، سنضيف بنية جديدة فارغة:

```
pub struct Emu {
}
```

هذه البنية ستكون الكائن الرئيسي لواجهة المحاكي الخلفية، وبالتالي يجب أن تتعامل مع تشغيل اللعبة الفعلية، وأن تكون قادرة على تمرير المعلومات المطلوبة ذهابًا وإيابًا من الواجهة الأمامية (مثل ما هو موجود على الشاشة وضغطات الأزرار).

عداد البرنامج

ولكن ماذا نضع في كائن `Emu` الخاص بنا؟ كما ناقشنا سابقًا، يحتاج البرنامج إلى معرفة مكان التنفيذ الحالي في اللعبة. تقوم جميع وحدات المعالجة المركزية بتحقيق ذلك عن طريق الاحتفاظ بفهرس للتعليمة الحالية، مخزنة في *سجل* خاص يعرف باسم *عداد البرنامج*، أو PC اختصارًا. سيكون هذا مفتاحًا لجزء *الجلب* من الحلقة، وسيزداد خلال اللعبة أثناء تشغيلها، ويمكن حتى تعديله يدويًا بواسطة بعض التعليمات (لأشياء مثل القفز إلى قسم مختلف من الكود أو استدعاء subroutine). دعنا نضيف هذا إلى البنية:

```
pub struct Emu {
        pc: u16,
        }
```

الذاكرة العشوائية (RAM)

بينما يمكننا القراءة من ملف اللعبة في كل مرة نحتاج فيها إلى تعليمة جديدة، إلا أن هذا بطيء جدًا وغير فعال، وببساطة ليس كيف تعمل الأنظمة الحقيقية. بدلاً من ذلك، تم تصميم Chip-8 لنسخ برنامج اللعبة بالكامل في مساحة الذاكرة

العشوائية الخاصة به، حيث يمكن بعد ذلك القراءة منه والكتابة إليه حسب الحاجة. تجدر الإشارة إلى أن العديد من الأنظمة، مثل Game Boy، لا تسمح لوحدة المعالجة المركزية بالكتابة فوق مناطق الذاكرة التي يتم تخزين اللعبة فيها (لن ترغب في أن يتسبب خطأ في إفساد كود اللعبة بالكامل). ومع ذلك، فإن Chip-8 لا يحتوي على مثل هذا القيد. نظرًا لأن Chip-8 لم يكن نظامًا ماديًا أبدًا، فلا يوجد معيار رسمي لكمية الذاكرة التي يجب أن يحتويها. ومع ذلك، فقد تم تصميمه في الأصل ليتم تنفيذه على أجهزة كمبيوتر تحتوي على 4096 بايت (4 كيلوبايت) من الذاكرة العشوائية، لذا هذا ما سنمنحه أيضًا. لن تستخدم معظم البرامج كل هذه الذاكرة، لكنها موجودة إذا احتاجت إليها. دعنا نحدد ذلك في برنامجنا.

```
const RAM_SIZE: usize = 4096;

pub struct Emu {
    pc: u16,
    ram: [u8; RAM_SIZE],
}
```

الشاشة

يستخدم Chip-8 شاشة أحادية اللون بدقة 64x32 (1 بت لكل بكسل). لا يوجد شيء خاص جدًا في هذا، ومع ذلك فهو أحد الأشياء القليلة في واجهتنا الخلفية التي يجب أن تكون قابلة للوصول من قبل واجهاتنا الأمامية المختلفة، وللمستخدم. على عكس العديد من الأنظمة، لا يقوم Chip-8 بمسح شاشته تلقائيًا لإعادة الرسم في كل إطار، بدلاً من ذلك يتم الحفاظ على حالة الشاشة،

ويتم رسم الرسومات الجديدة عليها (هناك أمر لمسح الشاشة). يمكننا الاحتفاظ ببيانات الشاشة هذه في مصفوفة في كائن Emu الخاص بنا. يعتبر Chip-8 أيضًا أكثر أساسية من معظم الأنظمة حيث أننا نتعامل فقط مع لونين - الأسود والأبيض. نظرًا لأن هذه شاشة 1 بت، يمكننا ببساطة تخزين مصفوفة من القيم المنطقية كما يلي:

```
pub const SCREEN_WIDTH: usize = 64;
pub const SCREEN_HEIGHT: usize = 32;

const RAM_SIZE: usize = 4096;

pub struct Emu {
    pc: u16,
    ram: [u8; RAM_SIZE],
    screen: [bool; SCREEN_WIDTH * SCREEN_HEIGHT],
}
```

ستلاحظ أيضًا أنه على عكس الثابت السابق، قمنا بتعريف أبعاد الشاشة كثوابت عامة. هذه واحدة من القطع القليلة من المعلومات التي ستكون الواجهة الأمامية بحاجة إليها لرسم الشاشة فعليًا.

سجلات V

بينما يحتوي النظام على قدر كبير من الذاكرة العشوائية للعمل بها، يعتبر الوصول إلى الذاكرة العشوائية بطيئًا نسبيًا (ولكنه لا يزال أسرع بكثير من القراءة من القرص). لتسريع الأمور، يعرّف Chip-8 ستة عشر سجلًا من 8 بت يمكن للعبة استخدامها كما تشاء لعمليات أسرع. يُشار إلى هذه السجلات باسم *سجلات V*،

وعادة ما يتم ترقيمها بالنظام الست عشري من V0 إلى VF (لست متأكدًا تمامًا مما يعنيه الحرف *V*)، وسنقوم بتجميعها معًا في مصفوفة واحدة في بنية Emu الخاصة بنا.

```
pub const SCREEN_WIDTH: usize = 64;
pub const SCREEN_HEIGHT: usize = 32;

const RAM_SIZE: usize = 4096;
const NUM_REGS: usize = 16;

pub struct Emu {
    pc: u16,
    ram: [u8; RAM_SIZE],
    screen: [bool; SCREEN_WIDTH * SCREEN_HEIGHT],
    v_reg: [u8; NUM_REGS],
}
```

سجل I

هناك أيضًا سجل آخر من 16 بت يعرف باسم *سجل I*، والذي يستخدم للفهرسة في الذاكرة العشوائية للقراءة والكتابة. سنتعمق في التفاصيل الدقيقة لكيفية عمل هذا لاحقًا، ولكن في الوقت الحالي نحتاج فقط إلى وجوده.

```
pub const SCREEN_WIDTH: usize = 64;
pub const SCREEN_HEIGHT: usize = 32;

const RAM_SIZE: usize = 4096;
const NUM_REGS: usize = 16;

pub struct Emu {
    pc: u16,
    ram: [u8; RAM_SIZE],
    screen: [bool; SCREEN_WIDTH * SCREEN_HEIGHT],
    v_reg: [u8; NUM_REGS],
```

```
    i_reg: u16,
}
```

المكدس

تحتوي وحدة المعالجة المركزية أيضًا على *مكدس* صغير، وهو عبارة عن مصفوفة من القيم ذات 16 بت يمكن لوحدة المعالجة المركزية القراءة منها والكتابة إليها. يختلف المكدس عن الذاكرة العشوائية العادية حيث يمكن فقط القراءة/الكتابة من/إلى المكدس عبر مبدأ "آخر داخل، أول خارج (LIFO)" (مثل كومة من الفطائر!)، عندما تذهب لالتقاط (إخراج) قيمة، فإنك تزيل آخر قيمة قمت بإضافتها (إدخالها). على عكس العديد من الأنظمة، لا يُستخدم المكدس لأغراض عامة. الأوقات الوحيدة المسموح فيها باستخدام المكدس هي عند الدخول أو الخروج من subroutine، حيث يتم استخدام المكدس لمعرفة المكان الذي بدأت منه حتى تتمكن من العودة بعد انتهاء الروتين. مرة أخرى، لا يحدد Chip-8 رسميًا عدد الأرقام التي يمكن أن يحتفظ بها المكدس، ولكن 16 هو رقم شائع لمطوري المحاكاة. هناك عدد من الطرق المختلفة التي يمكننا من خلالها تنفيذ المكدس الخاص بنا، ربما تكون أسهل طريقة هي استخدام الكائن `std::collections::VecDeque` من مكتبة Rust القياسية. يعمل هذا بشكل جيد لبناء سطح المكتب فقط، ولكن في وقت كتابة هذا المقال، لا تعمل العديد من العناصر في مكتبة `std` مع إصدارات WebAssembly. بدلاً من ذلك، سنقوم بذلك بالطريقة القديمة، باستخدام مصفوفة ذات حجم ثابت وفهرس حتى نعرف مكان الجزء العلوي من المكدس، والمعروف باسم *مؤشر المكدس* (SP).

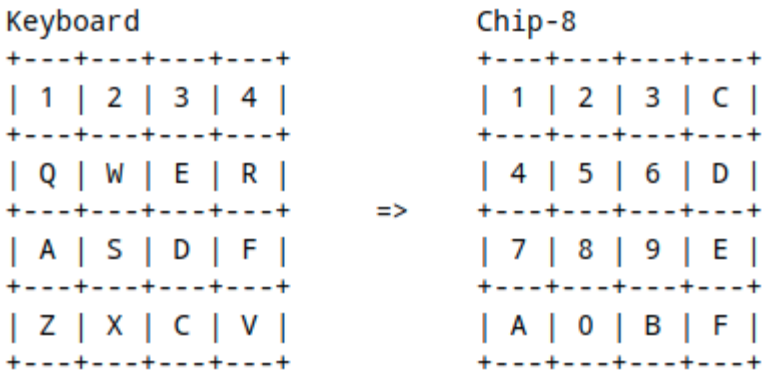
```rust
pub const SCREEN_WIDTH: usize = 64;
pub const SCREEN_HEIGHT: usize = 32;

const RAM_SIZE: usize = 4096;
const NUM_REGS: usize = 16;
const STACK_SIZE: usize = 16;

pub struct Emu {
    pc: u16,
    ram: [u8; RAM_SIZE],
    screen: [bool; SCREEN_WIDTH * SCREEN_HEIGHT],
    v_reg: [u8; NUM_REGS],
    i_reg: u16,
    sp: u16,
    stack: [u16; STACK_SIZE],
}
```

الأزرار

يدعم Chip-8 عددًا كبيرًا بشكل مدهش من الأزرار يصل إلى 16 زرًا، وعادة ما يتم ترقيمها بالنظام الست عشري من 0 إلى 9، ومن A إلى F. يتم ترتيب الأزرار بشكل مشابه لتخطيط الهاتف، مع وضع A و B على جانبي 0، و C إلى F على العمود الأيمن، مما يشكل شبكة 4x4.



تخطيط لوحة المفاتيح إلى أزرار Chip-8

نحتاج إلى تتبع الأزرار التي يتم الضغط عليها، وبالتالي يمكننا استخدام مصفوفة من القيم المنطقية لتتبع الحالات.

```
pub const SCREEN_WIDTH: usize = 64;
pub const SCREEN_HEIGHT: usize = 32;

const RAM_SIZE: usize = 4096;
const NUM_REGS: usize = 16;
const STACK_SIZE: usize = 16;
const NUM_KEYS: usize = 16;

pub struct Emu {
    pc: u16,
    ram: [u8; RAM_SIZE],
    screen: [bool; SCREEN_WIDTH * SCREEN_HEIGHT],
    v_reg: [u8; NUM_REGS],
    i_reg: u16,
    sp: u16,
    stack: [u16; STACK_SIZE],
    keys: [bool; NUM_KEYS],
}
```

المؤقتات

لقد كان هذا كثيرًا للتعامل معه دفعة واحدة، ولكننا الآن في العناصر النهائية. يحتوي Chip-8 أيضًا على سجلين خاصين آخرين يستخدمهما كمؤقتات. الأول، *مؤقت التأخير*، يستخدمه النظام كمؤقت عادي، حيث يقوم بالعد التنازلي في كل دورة ويقوم ببعض الإجراءات إذا وصل إلى 0. أما *مؤقت الصوت*، فيقوم أيضًا بالعد التنازلي في كل دورة ساعة، ولكن عند الوصول إلى 0 يصدر صوتًا. تعيين *مؤقت الصوت* إلى 0 هو الطريقة الوحيدة لإصدار الصوت على Chip-8، كما

سنرى لاحقًا. كلاهما عبارة عن سجلات 8 بت، ويجب دعمهما حتى نتمكن من المتابعة.

```
pub const SCREEN_WIDTH: usize = 64;
pub const SCREEN_HEIGHT: usize = 32;

const RAM_SIZE: usize = 4096;
const NUM_REGS: usize = 16;
const STACK_SIZE: usize = 16;
const NUM_KEYS: usize = 16;

pub struct Emu {
    pc: u16,
    ram: [u8; RAM_SIZE],
    screen: [bool; SCREEN_WIDTH * SCREEN_HEIGHT],
    v_reg: [u8; NUM_REGS],
    i_reg: u16,
    sp: u16,
    stack: [u16; STACK_SIZE],
    keys: [bool; NUM_KEYS],
    dt: u8,
    st: u8,
}
```

التهيئة

هذا يكفي في الوقت الحالي، دعنا ننفذ مُنشئ new لهذه الفئة قبل أن ننتقل إلى الجزء التالي. بعد تعريف struct، سنقوم بتنفيذ وتعيين القيم الافتراضية:

```
// -- الكود غير المتغير تم حذفه --

const START_ADDR: u16 = 0x200;

// -- الكود غير المتغير تم حذفه --

impl Emu {
    pub fn new() -> Self {
```

```
            Self {
                pc: START_ADDR,
                ram: [0; RAM_SIZE],
                screen: [false; SCREEN_WIDTH * SCREEN_HEIGHT],
                v_reg: [0; NUM_REGS],
                i_reg: 0,
                sp: 0,
                stack: [0; STACK_SIZE],
                keys: [false; NUM_KEYS],
                dt: 0,
                st: 0,
            }
        }
    }
```

يبدو كل شيء واضحًا إلى حد ما، نقوم ببساطة بتهيئة جميع القيم والمصفوفات إلى الصفر... باستثناء عداد البرنامج (Program Counter)، الذي يتم تعيينه إلى 0x200 (أي 512 بالنظام العشري). لقد ذكرت السبب وراء ذلك في الفصل السابق، ولكن يجب على المحاكي أن يعرف مكان بداية البرنامج، ومن المعيار في نظام Chip-8 أن يتم تحميل بداية جميع برامج Chip-8 من عنوان الذاكرة 0x200. هذا الرقم سيظهر مرة أخرى، لذا قمنا بتعريفه كقيمة ثابتة.

بهذا نكون قد انتهينا من هذا الجزء! مع وضع الأساس لمحاكاتنا، يمكننا البدء في تنفيذ عملية التنفيذ!

تنفيذ طرق المحاكاة

لقد أنشأنا الآن بنية `Emu` وحددنا عددًا من المتغيرات لها لإدارتها، بالإضافة إلى تحديد وظيفة التهيئة. قبل أن ننتقل، هناك بعض الطرق المفيدة التي يجب أن نضيفها إلى كائننا الآن والتي ستكون مفيدة بمجرد أن نبدأ في تنفيذ التعليمات.

Push و Pop

لقد أضفنا كلًا من مصفوفة `stack` ومؤشر `sp` لإدارة مكدس وحدة المعالجة المركزية، ومع ذلك سيكون من المفيد تنفيذ كل من وظيفتي `push` و `pop` حتى نتمكن من الوصول إليها بسهولة.

```
impl Emu {
// -- كود غير متغير محذوف --

    fn push(&mut self, val: u16) {
self.stack[self.sp as usize] = val;
        self.sp += 1;
    }

    fn pop(&mut self) -> u16 {
        self.sp -= 1;
self.stack[self.sp as usize]
    }

// -- كود غير متغير محذوف --
}
```

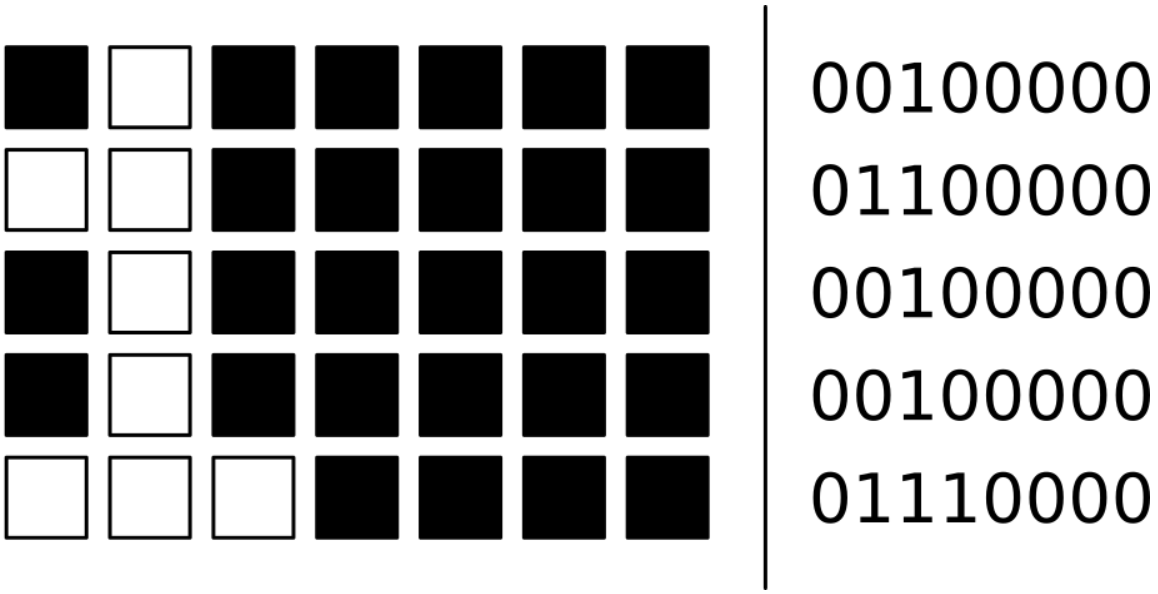
هذه الوظائف واضحة جدًا. `push` تضيف القيمة المعطاة التي تبلغ 16 بت إلى المكان الذي يشير إليه مؤشر المكدس، ثم تحرك المؤشر إلى الموضع التالي. `pop`

تنفذ هذه العملية بشكل عكسي، حيث تحرك مؤشر المكدس إلى القيمة السابقة ثم تعيد ما يوجد هناك. لاحظ أن محاولة إجراء pop على مكدس فارغ يؤدي إلى حدوث خطأ تحت التدفق (underflow panic)1. يمكنك إضافة معالجة إضافية هنا إذا أردت، ولكن في حالة حدوث ذلك، فإن ذلك يشير إلى وجود خلل في المحاكي أو كود اللعبة، لذلك أعتقد أن حدوث خطأ كامل مقبول.

خطوط الرموز (Font Sprites)

لم نتعمق بعد في كيفية عمل شاشة عرض Chip-8، ولكن الفكرة الأساسية الآن هي أنها تعرض *الرموز* (sprites) التي يتم تخزينها في الذاكرة على الشاشة، سطرًا واحدًا في كل مرة. يقع على عاتق مطور اللعبة تحميل الرموز بشكل صحيح قبل نسخها إلى الشاشة. ولكن ألن يكون من الجميل إذا كان النظام يحتوي تلقائيًا على رموز للأشياء الشائعة الاستخدام، مثل الأرقام؟ لقد ذكرت سابقًا أن عداد البرنامج (PC) سيبدأ من العنوان 0x200، تاركًا أول 512 بايت فارغة عن قصد. تستخدم معظم المحاكيات الحديثة هذا المساحة لتخزين بيانات الرموز لأحرف الخط الخاصة بجميع الأرقام السداسية عشرية، أي الأحرف من 0-9 و A-F. يمكننا تخزين هذه البيانات في أي موقع ثابت في الذاكرة العشوائية (RAM)، ولكن هذه المساحة محددة بالفعل على أنها فارغة على أي حال. يتكون كل حرف من خمسة صفوف من ثمانية وحدات بكسل، ويستخدم كل صف بايت واحد من البيانات، مما يعني أن كل حرف يستغرق خمسة بايتات من البيانات بشكل كامل. يوضح الرسم التالي كيفية تخزين الحرف كبايتات.

2 *التدفق تحت الصفر* (Underflow) هو عندما تتحول قيمة متغير غير موقعة من فوق الصفر إلى تحت الصفر. في بعض اللغات، قد "تدور" القيمة إلى أعلى قيمة ممكنة، ولكن في Rust يؤدي هذا إلى حدوث خطأ في وقت التشغيل ويجب التعامل معه بشكل مختلف إذا كان مطلوبًا. الأمر نفسه ينطبق على القيم التي تتجاوز القيمة القصوى الممكنة، والمعروفة باسم *التدفق فوق الصفر* (overflow).



رموز خط Chip-8

على اليمين، يتم ترميز كل صف في شكل ثنائي. يتم تعيين كل بكسل إلى بت، والذي يتوافق مع ما إذا كان هذا البكسل سيكون أبيض أو أسود. *كل* الرموز في Chip-8 بعرض ثمانية وحدات بكسل، مما يعني أن صف البكسل يتطلب 8 بت (1 بايت). يوضح الرسم أعلاه تخطيط رمز الحرف "1". لا تحتاج الرموز إلى جميع وحدات البت الثمانية من العرض، لذا فإن جميعها تحتوي على نصف أيمن أسود. تم إنشاء رموز لجميع الأرقام السداسية عشرية، ويجب أن تكون موجودة

في مكان ما في الذاكرة العشوائية لكي تعمل بعض الألعاب. لاحقًا في هذا الدليل سنغطي التعليمات التي تتعامل مع هذه الرموز، والتي ستوضح كيفية تحميلها وكيف يعرف المحاكي مكان العثور عليها. في الوقت الحالي، نحتاج ببساطة إلى تعريفها. سنفعل ذلك باستخدام مصفوفة ثابتة من البايتات؛ في أعلى ملف `lib.rs`، أضف:

```
const FONTSET_SIZE: usize = 80;

const FONTSET: [u8; FONTSET_SIZE] = [
    0xF0, 0x90, 0x90, 0x90, 0xF0, // 0
    0x20, 0x60, 0x20, 0x20, 0x70, // 1
    0xF0, 0x10, 0xF0, 0x80, 0xF0, // 2
    0xF0, 0x10, 0xF0, 0x10, 0xF0, // 3
    0x90, 0x90, 0xF0, 0x10, 0x10, // 4
    0xF0, 0x80, 0xF0, 0x10, 0xF0, // 5
    0xF0, 0x80, 0xF0, 0x90, 0xF0, // 6
    0xF0, 0x10, 0x20, 0x40, 0x40, // 7
    0xF0, 0x90, 0xF0, 0x90, 0xF0, // 8
    0xF0, 0x90, 0xF0, 0x10, 0xF0, // 9
    0xF0, 0x90, 0xF0, 0x90, 0x90, // A
    0xE0, 0x90, 0xE0, 0x90, 0xE0, // B
    0xF0, 0x80, 0x80, 0x80, 0xF0, // C
    0xE0, 0x90, 0x90, 0x90, 0xE0, // D
    0xF0, 0x80, 0xF0, 0x80, 0xF0, // E
    0xF0, 0x80, 0xF0, 0x80, 0x80  // F
                                ];
```

يمكنك رؤية البايتات الموضحة في الرسم البياني للحرف "1" أعلاه، وجميع الأحرف الأخرى تعمل بطريقة مماثلة. الآن بعد أن تم تحديد هذه البايتات، نحتاج إلى تحميلها في الذاكرة العشوائية. قم بتعديل `Emu::new()` لنسخ هذه القيم:

```
    pub fn new() -> Self {
        let mut new_emu = Self {
            pc: START_ADDR,
```

```
            ram: [0; RAM_SIZE],
        screen: [false; SCREEN_WIDTH * SCREEN_HEIGHT],
            v_reg: [0; NUM_REGS],
                i_reg: 0,
                   sp: 0,
             stack: [0; STACK_SIZE],
            keys: [false; NUM_KEYS],
                   dt: 0,
                   st: 0,
                       };

    new_emu.ram[..FONTSET_SIZE].copy_from_slice(&FONTSET);

                    new_emu
                        }
```

يقوم هذا بتهيئة كائن `Emu` بنفس الطريقة كما كان من قبل، ولكن ينسخ بيانات رموز الأحرف إلى الذاكرة العشوائية قبل إرجاعه.

سيكون من المفيد أيضًا أن نتمكن من إعادة تعيين المحاكي دون الحاجة إلى إنشاء كائن جديد. هناك طرق أكثر تطورًا للقيام بذلك، ولكننا سنبقيها بسيطة وننشئ وظيفة تعيد تعيين متغيرات العضو إلى قيمها الأصلية عند استدعائها.

```
                pub fn reset(&mut self) {
                    self.pc = START_ADDR;
                self.ram = [0; RAM_SIZE];
    self.screen = [false; SCREEN_WIDTH * SCREEN_HEIGHT];
                self.v_reg = [0; NUM_REGS];
                        self.i_reg = 0;
                           self.sp = 0;
                 self.stack = [0; STACK_SIZE];
                self.keys = [false; NUM_KEYS];
                           self.dt = 0;
                           self.st = 0;
     self.ram[..FONTSET_SIZE].copy_from_slice(&FONTSET);
                                }
```

Tick

مع اكتمال إنشاء كائن `Emu` (في الوقت الحالي)، يمكننا البدء في تحديد كيفية معالجة وحدة المعالجة المركزية لكل تعليمة والتحرك خلال اللعبة. لتلخيص ما تم وصفه في الأجزاء السابقة، ستكون الحلقة الأساسية كما يلي:

1. جلب القيمة من لعبتنا (التي تم تحميلها في الذاكرة العشوائية) من عنوان الذاكرة المخزن في عداد البرنامج (Program Counter).
2. فك تشفير هذه التعليمات.
3. التنفيذ، والذي قد يتضمن تعديل سجلات وحدة المعالجة المركزية أو الذاكرة العشوائية.
4. تحريك عداد البرنامج إلى التعليمات التالية وتكرار العملية.

لنبدأ بإضافة معالجة الأوامر إلى وظيفة `tick`، بدءًا من خطوة الجلب:

```
// -- كود غير متغير محذوف --

pub fn tick(&mut self) {
    // الجلب
    let op = self.fetch();
    // فك التشفير
    // التنفيذ
}

fn fetch(&mut self) -> u16 {
    // TODO
}
```

سيتم استدعاء وظيفة `fetch` داخليًا فقط كجزء من حلقة `tick`، لذا لا تحتاج إلى أن تكون عامة. الغرض من هذه الوظيفة هو الحصول على التعليمات التي نحن على وشك تنفيذها (المعروفة باسم *opcode*) لاستخدامها في الخطوات التالية من هذه الدورة. إذا لم تكن على دراية بتنسيق تعليمات Chip-8، أوصي بمراجعة الملخص من الفصول السابقة.

لحسن الحظ، فإن Chip-8 أسهل من العديد من الأنظمة. أولاً، هناك فقط 35 opcode للتعامل معها مقارنة بالمئات التي تدعمها العديد من المعالجات. بالإضافة إلى ذلك، تقوم العديد من الأنظمة بتخزين معلمات إضافية لكل opcode في البايتات اللاحقة (مثل المعاملات للإضافة)، بينما يقوم Chip-8 بترميز هذه المعلمات في الـ opcode نفسه. بسبب هذا، فإن جميع opcodes في Chip-8 هي بالضبط 2 بايت، وهو أكبر من بعض الأنظمة الأخرى، ولكن التعليمات بأكملها مخزنة في هذين البايتين، بينما قد تستهلك الأنظمة الأخرى المعاصرة ما بين 1 إلى 3 بايتات لكل دورة.

يتم ترميز كل opcode بشكل مختلف، ولكن لحسن الحظ نظرًا لأن جميع التعليمات تستهلك بايتين، فإن عملية الجلب هي نفسها لجميعها، ويتم تنفيذها على النحو التالي:

```
                                fn fetch(&mut self) -> u16 {
        let higher_byte = self.ram[self.pc as usize] as u16;
    let lower_byte = self.ram[(self.pc + 1) as usize] as u16;
                    let op = (higher_byte << 8) | lower_byte;
                                                self.pc += 2;
```

```
op
}
```

تقوم هذه الوظيفة بجلب الـ opcode الذي يبلغ 16 بت المخزن في عداد البرنامج الحالي. نقوم بتخزين القيم في الذاكرة العشوائية كقيم 8 بت، لذا نقوم بجلب اثنين ودمجهما كـ Big Endian. ثم يتم زيادة عداد البرنامج بمقدار البايتين اللذين قرأناهما للتو، ويتم إرجاع الـ opcode الذي تم جلبه لمزيد من المعالجة.

Timer Tick

تذكر مواصفات Chip-8 أيضًا وجود مؤقتين خاصين، مؤقت التأخير (Delay Timer) ومؤقت الصوت (Sound Timer). بينما تعمل وظيفة `tick` مرة واحدة في كل دورة لوحدة المعالجة المركزية، يتم تعديل هذه المؤقتات بدلاً من ذلك مرة واحدة في كل إطار، وبالتالي يجب التعامل معها في وظيفة منفصلة. سلوكها بسيط للغاية، ففي كل إطار ينخفض كل منهما بمقدار واحد. إذا تم تعيين مؤقت الصوت إلى واحد، فإن النظام سيصدر صوت "بيب". إذا وصلت المؤقتات إلى الصفر، فإنها لا تعيد تعيين نفسها تلقائيًا؛ بل تبقى عند الصفر حتى تقوم اللعبة بإعادة تعيينها يدويًا إلى بعض القيم.

```rust
pub fn tick_timers(&mut self) {
    if self.dt > 0 {
        self.dt -= 1;
    }

    if self.st > 0 {
        if self.st == 1 {
            // BEEP
        }
```

```
            self.st -= 1;
        }
    }
```

الصوت هو الشيء الوحيد الذي لن يغطيه هذا الدليل، وذلك بسبب التعقيد المتزايد في جعل الصوت يعمل في كل من واجهاتنا الأمامية لسطح المكتب ومتصفح الويب. في الوقت الحالي سنترك ببساطة تعليقًا حيث سيحدث الصوت، ولكن أي قراء فضوليين مدعوون لتنفيذ ذلك بأنفسهم (ثم إخباري كيف فعلوا ذلك).

---

1. ‘ROM’ تعني “ذاكرة القراءة فقط” (Read-only memory). وهي نوع من الذاكرة يتم كتابتها بشكل دائم أثناء التصنيع ولا يمكن تعديلها بواسطة نظام الكمبيوتر. لأغراض هذا الدليل، سيتم استخدام “ملف ROM” بالتبادل مع “ملف اللعبة”.

2. ‘ROM’ تعني “ذاكرة القراءة فقط” (Read-only memory). وهي نوع من الذاكرة يتم كتابتها بشكل دائم أثناء التصنيع ولا يمكن تعديلها بواسطة نظام الكمبيوتر. لأغراض هذا الدليل، سيتم استخدام “ملف ROM” بالتبادل مع “ملف اللعبة”.

تنفيذ الأوامر (Opcode Execution)

في القسم السابق، قمنا بجلب الـ opcode وكنا نستعد لفك تشفير التعليمات التي يتوافق معها لتنفيذ تلك التعليمات. حاليًا، تبدو وظيفة `tick` كما يلي:

```
pub fn tick(&mut self) {
                    // الجلب
    let op = self.fetch();
                    // فك التشفير
                    // التنفيذ
                }
```

هذا يعني أن فك التشفير والتنفيذ سيكونان وظيفتين منفصلتين. بينما يمكن أن يكونا كذلك، بالنسبة لـ Chip-8، من الأسهل ببساطة تنفيذ العملية أثناء تحديدها، بدلاً من تضمين استدعاء وظيفة أخرى. وبالتالي تصبح وظيفة `tick` كما يلي:

```
pub fn tick(&mut self) {
                    // الجلب
    let op = self.fetch();
                // فك التشفير والتنفيذ
                self.execute(op);
                }

fn execute(&mut self, op: u16) {
                    // TODO
                }
```

خطوتنا التالية هي *فك التشفير*، أو تحديد بالضبط أي عملية نتعامل معها. تحتوي ورقة الغش الخاصة بأوامر Chip-8 على جميع الأوامر المتاحة، وكيفية تفسير

معاملاتها، وبعض الملاحظات حول معناها. ستحتاج إلى الرجوع إليها كثيرًا.

بالنسبة لمحاكي كامل، يجب تنفيذ كل واحدة منها.

مطابقة الأنماط (Pattern Matching)

لحسن الحظ، تمتلك Rust ميزة مطابقة الأنماط القوية والمفيدة التي يمكننا استخدامها لصالحنا. ومع ذلك، سنحتاج إلى فصل كل "رقم" سداسي عشري قبل القيام بذلك.

```
 fn execute(&mut self, op: u16) {
let digit1 = (op & 0xF000) >> 12;
 let digit2 = (op & 0x0F00) >> 8;
 let digit3 = (op & 0x00F0) >> 4;
        let digit4 = op & 0x000F;
                                }
```

ربما ليس هذا الكود الأكثر نظافة، ولكننا نحتاج إلى كل رقم سداسي عشري بشكل منفصل. من هنا، يمكننا إنشاء عبارة `match` حيث يمكننا تحديد الأنماط لجميع أوامرنا:

```
 fn execute(&mut self, op: u16) {
let digit1 = (op & 0xF000) >> 12;
 let digit2 = (op & 0x0F00) >> 8;
 let digit3 = (op & 0x00F0) >> 4;
        let digit4 = op & 0x000F;

    match (digit1, digit2, digit3, digit4) {
(_, _, _, _) => unimplemented!("Unimplemented opcode: {}
                                }
                                }
```

تتطلب عبارة `match` في Rust أن يتم أخذ جميع الخيارات الممكنة في الاعتبار، وهو ما يتم باستخدام المتغير `_`، الذي يلتقط "كل شيء آخر". في الداخل، سنستخدم الماكرو `unimplemented!` للتسبب في ذعر البرنامج إذا وصل إلى هذه النقطة. بحلول الوقت الذي ننتهي فيه من إضافة جميع الأوامر، يطالب مترجم Rust بأن يكون لدينا عبارة "كل شيء آخر"، ولكننا لا ينبغي أن نصل إليها أبدًا.

بينما يمكن أن تعمل عبارة `match` الطويلة بالتأكيد مع بنى أخرى، فمن الشائع أكثر تنفيذ التعليمات في وظائفها الخاصة، واستخدام جدول بحث أو تحديد الوظيفة الصحيحة برمجيًا. Chip-8 غير عادية بعض الشيء لأنها تخزن معلمات التعليمات في الـ opcode نفسه، مما يعني أننا نحتاج إلى الكثير من الرموز البديلة لمطابقة التعليمات. نظرًا لوجود عدد صغير نسبيًا منها، تعمل عبارة `match` بشكل جيد هنا.

مع إعداد الإطار، دعنا نبدأ!

مقدمة لتنفيذ الأوامر

تناقش الصفحات التالية كيفية عمل جميع تعليمات Chip-8، وتتضمن كودًا لكيفية تنفيذها. أنت مرحب ببساطة باتباعها وتنفيذ التعليمات واحدة تلو الأخرى، ولكن قبل أن تفعل ذلك، قد ترغب في النظر إلى القسم التالي والبدء في العمل على بعض كود الواجهة الأمامية. حاليًا، ليس لدينا طريقة لتشغيل المحاكي فعليًا، وقد يكون من المفيد لبعض الأشخاص محاولة تحميل وتشغيل لعبة

لأغراض التصحيح. ومع ذلك، تذكر أن المحاكي من المحتمل أن يتعطل بسرعة كبيرة ما لم يتم تنفيذ جميع التعليمات. شخصيًا، أفضل العمل على التعليمات أولاً قبل العمل على الأجزاء المتحركة الأخرى (ولهذا تم وضع هذا الدليل بهذه الطريقة).

مع هذا التحذير، دعنا ننتقل إلى العمل على كل تعليمات Chip-8 بالترتيب.

Nop - 0000

أول تعليمة لدينا هي الأبسط - لا تفعل شيئًا. قد يبدو هذا سخيفًا، ولكن في بعض الأحيان يكون ضروريًا لأغراض التوقيت أو المحاذاة. على أي حال، نحتاج ببساطة إلى الانتقال إلى الـ opcode التالي (الذي تم بالفعل في `fetch`)، والعودة.

```
                match (digit1, digit2, digit3, digit4) {
                                                // NOP
                                        (0, 0, 0, 0) => return,
(_, _, _, _) => unimplemented!("Unimplemented opcode: {}", o
                                                }
```

00E0 - مسح الشاشة

الـ opcode 0x00E0 هو التعليمات لمسح الشاشة، مما يعني أننا بحاجة إلى إعادة تعيين مخزن الشاشة ليكون فارغًا مرة أخرى.

```
                match (digit1, digit2, digit3, digit4) {
                                        // -- كود غير متغير محذوف --

                                                // CLS
                                        (0, 0, 0xE, 0) => {
self.screen = [false; SCREEN_WIDTH * SCREEN_HEIGHT];
                                                },
```

```
// -- كود غير متغير محذوف --
}
```

00EE - العودة من الروتين الفرعي

لم نتحدث بعد عن الروتينات الفرعية (المعروفة أيضًا باسم الوظائف) وكيف تعمل. الدخول إلى روتين فرعي يعمل بنفس طريقة القفز العادي؛ ننقل عداد البرنامج (PC) إلى العنوان المحدد ونستأنف التنفيذ من هناك. على عكس القفز، من المتوقع أن يكمل الروتين الفرعي في مرحلة ما، وسنحتاج إلى العودة إلى النقطة التي دخلنا منها. هذا هو المكان الذي يأتي فيه المكدس. عندما ندخل إلى روتين فرعي، ندفع ببساطة عنواننا إلى المكدس، ونشغل كود الروتين، وعندما نكون مستعدين للعودة، نخرج هذه القيمة من المكدس وننفذ من تلك النقطة مرة أخرى. يسمح لنا المكدس أيضًا بالحفاظ على عناوين العودة للروتينات الفرعية المتداخلة مع ضمان إعادتها بالترتيب الصحيح.

```
match (digit1, digit2, digit3, digit4) {
// -- كود غير متغير محذوف --

                    // RET
        (0, 0, 0xE, 0xE) => {
    let ret_addr = self.pop();
        self.pc = ret_addr;
                        },

// -- كود غير متغير محذوف --
                    }
```

1NNN - القفز

تعليمة القفز سهلة الإضافة، ببساطة انقل عداد البرنامج (PC) إلى العنوان المحدد.

```
match (digit1, digit2, digit3, digit4) {
// -- كود غير متغير محذوف --

// JMP NNN
(1, _, _, _) => {
let nnn = op & 0xFFF;
self.pc = nnn;
},

// -- كود غير متغير محذوف --
}
```

الشيء الرئيسي الذي يجب ملاحظته هنا هو أن هذا الـ opcode يتم تعريفه بـ ‘0x1’ كأكثر رقم معنوي. الأرقام الأخرى تستخدم كمعاملات لهذه العملية، ومن هنا جاء الرمز _ في عبارة المطابقة، هنا نريد أي شيء يبدأ بـ 1، ولكن ينتهي بأي ثلاثة أرقام للدخول إلى هذه العبارة.

2NNN - استدعاء روتين فرعي

عكس عملية “العودة من الروتين الفرعي”، سنقوم بإضافة عداد البرنامج الحالي إلى المكدس، ثم القفز إلى العنوان المحدد. إذا قفزت مباشرة إلى هنا، أوصي بقراءة قسم *العودة* للحصول على سياق إضافي.

```
match (digit1, digit2, digit3, digit4) {
// -- كود غير متغير محذوف --

// CALL NNN
(2, _, _, _) => {
let nnn = op & 0xFFF;
```

```
                self.push(self.pc);
                self.pc = nnn;
                },

            // -- كود غير متغير محذوف --
            }
```

3XNN - تخطي التالي إذا كان VX == NN

هذا الـ opcode هو الأول من بين عدد قليل يتبع نمطًا مشابهًا. بالنسبة لأولئك الذين ليسوا على دراية بلغة التجميع، فإن القدرة على تخطي سطر تعطي وظيفة مشابهة لكتلة if-else. يمكننا إجراء مقارنة، وإذا كانت النتيجة صحيحة نذهب إلى تعليمة واحدة، وإذا كانت خاطئة نذهب إلى مكان آخر. هذا أيضًا هو أول opcode يستخدم أحد *سجلات V*. في هذه الحالة، يخبرنا الرقم الثاني أي سجل نستخدمه، بينما يوفر الرقمان الأخيران القيمة الخام.

```
        match (digit1, digit2, digit3, digit4) {
            // -- كود غير متغير محذوف --

                // SKIP VX == NN
                (3, _, _, _) => {
                let x = digit2 as usize;
                let nn = (op & 0xFF) as u8;
                if self.v_reg[x] == nn {
                    self.pc += 2;
                }
                },

            // -- كود غير متغير محذوف --
            }
```

يعمل التنفيذ على النحو التالي: نظرًا لأننا لدينا بالفعل الرقم الثاني محفوظ في متغير، سنعيد استخدامه لفهرس 'X' الخاص بنا، على الرغم من تحويله إلى

usize، حيث تتطلب Rust أن يتم فهرسة المصفوفات باستخدام متغير usize.

إذا كانت القيمة المخزنة في هذا السجل تساوي nn، فإننا نتخطى الـ opcode التالي، وهو نفس تخطي عداد البرنامج (PC) بمقدار بايتين.

4XNN - تخطي التالي إذا كان VX != NN!

هذا الـ opcode هو بالضبط نفس السابق، إلا أننا نتخطى إذا كانت القيم المقارنة غير متساوية.

```
match (digit1, digit2, digit3, digit4) {
// -- كود غير متغير محذوف --

// SKIP VX != NN
(4, _, _, _) => {
let x = digit2 as usize;
let nn = (op & 0xFF) as u8;
if self.v_reg[x] != nn {
self.pc += 2;
}
},

// -- كود غير متغير محذوف --
}
```

5XY0 - تخطي التالي إذا كان VX == VY

عملية مشابهة مرة أخرى، ولكننا الآن نستخدم الرقم الثالث للفهرسة في سجل *V* آخر. ستلاحظ أيضًا أن الرقم الأقل أهمية لا يستخدم في العملية. يتطلب هذا الـ opcode أن يكون 0.

```
match (digit1, digit2, digit3, digit4) {
// -- كود غير متغير محذوف --
```

```
                // SKIP VX == VY
                (5, _, _, 0) => {
                    let x = digit2 as usize;
                    let y = digit3 as usize;
                    if self.v_reg[x] == self.v_reg[y] {
                        self.pc += 2;
                    }
                },

                // -- كود غير متغير محذوف --
            }
```

6XNN - VX = NN

قم بتعيين سجل V المحدد بواسطة الرقم الثاني إلى القيمة المعطاة.

```
        match (digit1, digit2, digit3, digit4) {
            // -- كود غير متغير محذوف --

                // VX = NN
                (6, _, _, _) => {
                    let x = digit2 as usize;
                    let nn = (op & 0xFF) as u8;
                    self.v_reg[x] = nn;
                },

                // -- كود غير متغير محذوف --
            }
```

7XNN - VX += NN

تقوم هذه العملية بإضافة القيمة المعطاة إلى سجل VX. في حالة التدفق الزائد، سيتسبب Rust في ذعر، لذا نحتاج إلى استخدام طريقة مختلفة عن عامل الجمع المعتاد. لاحظ أيضًا أنه بينما يحتوي Chip-8 على علامة حمل (المزيد عن ذلك لاحقًا)، إلا أنها لا يتم تعديلها بواسطة هذه العملية.

```
        match (digit1, digit2, digit3, digit4) {
            // -- كود غير متغير محذوف --

            // VX += NN
            (7, _, _, _) => {
                let x = digit2 as usize;
                let nn = (op & 0xFF) as u8;
                self.v_reg[x] = self.v_reg[x].wrapping_add(nn);
            },

            // -- كود غير متغير محذوف --
        }
```

8XY0 - VX = VY

مثل عملية `VX = NN`، ولكن القيمة المصدر هي من سجل VY.

```
        match (digit1, digit2, digit3, digit4) {
            // -- كود غير متغير محذوف --

            // VX = VY
            (8, _, _, 0) => {
                let x = digit2 as usize;
                let y = digit3 as usize;
                self.v_reg[x] = self.v_reg[y];
            },

            // -- كود غير متغير محذوف --
        }
```

العمليات البتية - 8XY1, 8XY2, 8XY3

أكواد `8XY1`، `8XY2`، و `8XY3` هي جميعها وظائف متشابهة، لذا بدلاً من تكرار نفسي ثلاث مرات، سأقوم بتنفيذ عملية *OR*، وأسمح للقارئ بتنفيذ العمليتين الأخريين.

```
        match (digit1, digit2, digit3, digit4) {
            // -- كود غير متغير محذوف --
```

```
                // VX |= VY
                (8, _, _, 1) => {
                    let x = digit2 as usize;
                    let y = digit3 as usize;
                    self.v_reg[x] |= self.v_reg[y];
                },

                // -- كود غير متغير محذوف --
```

8XY4 - VX += VY

هذه العملية لها جانبين يجب ملاحظتهما. أولاً، هذه العملية لديها القدرة على التدفق الزائد، مما سيسبب ذعرًا في Rust إذا لم يتم التعامل معها بشكل صحيح. ثانيًا، هذه العملية هي الأولى التي تستخدم سجل VF كعلامة. لقد ذكرت ذلك سابقًا، ولكن بينما تكون أول 15 سجلًا من *V* للاستخدام العام، فإن السجل السادس عشر (0xF) يعمل أيضًا كـ *سجل العلامات*. سجلات العلامات شائعة في العديد من معالجات وحدة المعالجة المركزية؛ في حالة Chip-8، فإنه يخزن أيضًا *علامة الحمل*، وهي بشكل أساسي متغير خاص يلاحظ إذا كانت آخر عملية تطبيق أدت إلى تدفق زائد/تدفق تحت. هنا، إذا حدث تدفق زائد، نحتاج إلى تعيين VF ليكون 1، أو 0 إذا لم يحدث. مع أخذ هذين الجانبين في الاعتبار، سنستخدم سمة `overflowing_add` في Rust، والتي ستعيد مجموعة من كل من المجموع الملتف، بالإضافة إلى قيمة منطقية تشير إلى ما إذا كان قد حدث تدفق زائد.

```
            match (digit1, digit2, digit3, digit4) {
                // -- كود غير متغير محذوف --

                // VX += VY
                (8, _, _, 4) => {
                    let x = digit2 as usize;
```

```
                let y = digit3 as usize;

let (new_vx, carry) = self.v_reg[x].overflowing_add(self
                let new_vf = if carry { 1 } else { 0 };

                self.v_reg[x] = new_vx;
                self.v_reg[0xF] = new_vf;
                },

                // -- كود غير متغير محذوف --
                }
```

8XY5 - VX -= VY

هذه العملية مشابهة للعملية السابقة، ولكن مع الطرح بدلاً من الجمع. الفرق الرئيسي هو أن علم الحمل VF يعمل بشكل معاكس. في عملية الجمع، يتم تعيين العلم إلى 1 إذا حدث تجاوز، أما هنا في حالة الطرح، إذا حدث نقص، يتم تعيين العلم إلى 0، والعكس صحيح. سنستخدم هنا الدالة overflowing_sub.

```
            match (digit1, digit2, digit3, digit4) {
                // -- كود غير متغير محذوف --

                // VX -= VY
                (8, _, _, 5) => {
                let x = digit2 as usize;
                let y = digit3 as usize;

let (new_vx, borrow) = self.v_reg[x].overflowing_sub(sel
                let new_vf = if borrow { 0 } else { 1 };

                self.v_reg[x] = new_vx;
                self.v_reg[0xF] = new_vf;
                },

                // -- كود غير متغير محذوف --
                }
```

8XY6 - VX >>= 1

تقوم هذه العملية بإزاحة واحدة إلى اليمين للقيمة في VX، مع تخزين البت الذي تم إسقاطه في السجل VF. لسوء الحظ، لا يوجد عامل تشغيل مدمج في Rust لالتقاط البت المسقط، لذا سنقوم بذلك يدويًا.

```
match (digit1, digit2, digit3, digit4) {
// -- كود غير متغير محذوف --

// VX >>= 1
(8, _, _, 6) => {
let x = digit2 as usize;
let lsb = self.v_reg[x] & 1;
self.v_reg[x] >>= 1;
self.v_reg[0xF] = lsb;
},

// -- كود غير متغير محذوف --
}
```

8XY7 - VX = VY - VX

تعمل هذه العملية بنفس طريقة العملية السابقة VX -= VY، ولكن مع تبديل المعاملات.

```
match (digit1, digit2, digit3, digit4) {
// -- كود غير متغير محذوف --

// VX = VY - VX
(8, _, _, 7) => {
let x = digit2 as usize;
let y = digit3 as usize;

let (new_vx, borrow) = self.v_reg[y].overflowing_sub(sel
let new_vf = if borrow { 0 } else { 1 };
```

```
                self.v_reg[x] = new_vx;
            self.v_reg[0xF] = new_vf;
                                },

            // -- كود غير متغير محذوف --
                                }
```

8XYE - VX <<= 1

تشبه عملية الإزاحة إلى اليمين، ولكننا نخزن القيمة التي تم تجاوزها في سجل العلم.

```
        match (digit1, digit2, digit3, digit4) {
            // -- كود غير متغير محذوف --

                                // VX <<= 1
                            (8, _, _, 0xE) => {
                        let x = digit2 as usize;
            let msb = (self.v_reg[x] >> 7) & 1;
                            self.v_reg[x] <<= 1;
                        self.v_reg[0xF] = msb;
                                },

            // -- كود غير متغير محذوف --
                                }
```

9XY0 - تخطي إذا VX != VY

بعد الانتهاء من عمليات 0x8000، حان الوقت لإضافة عملية كانت مفقودة، وهي تخطي السطر التالي إذا كانت VX لا تساوي VY. هذا الكود مشابه لعملية 5XY0، ولكن مع عدم المساواة.

```
        match (digit1, digit2, digit3, digit4) {
            // -- كود غير متغير محذوف --

                                // تخطي إذا VX != VY
```

```
                (9, _, _, 0) => {
            let x = digit2 as usize;
            let y = digit3 as usize;
        if self.v_reg[x] != self.v_reg[y] {
                    self.pc += 2;
                    }
                    },

        // -- كود غير متغير محذوف --
                    }
```

ANNN - I = NNN

هذه هي العملية الأولى التي تستخدم *سجل I*، والذي سيتم استخدامه في عدة عمليات إضافية، بشكل أساسي كمؤشر عنوان إلى الذاكرة العشوائية. في هذه الحالة، نقوم ببساطة بتعيينه إلى القيمة 0xNNN المشفرة في هذه العملية.

```
match (digit1, digit2, digit3, digit4) {
// -- كود غير متغير محذوف --

            // I = NNN
        (0xA, _, _, _) => {
    let nnn = op & 0xFFF;
        self.i_reg = nnn;
                    },

// -- كود غير متغير محذوف --
                    }
```

BNNN - القفز إلى V0 + NNN

بينما استخدمت العمليات السابقة *سجل V* المحدد في العملية، فإن هذه العملية تستخدم دائمًا *سجل V0*. تقوم هذه العملية بنقل PC إلى مجموع القيمة المخزنة في *V0* والقيمة الخام 0xNNN المقدمة في العملية.

```
match (digit1, digit2, digit3, digit4) {
// -- كود غير متغير محذوف --

// القفز إلى V0 + NNN
(0xB, _, _, _) => {
let nnn = op & 0xFFF;
self.pc = (self.v_reg[0] as u16) + nnn;
},

// -- كود غير متغير محذوف --
}
```

CXNN - VX = rand() & NN

أخيرًا، شيء لكسر الرتابة! هذه العملية هي عملية توليد الأرقام العشوائية في Chip-8، مع لمسة خاصة، حيث يتم بعد ذلك تطبيق عملية AND على الرقم العشوائي مع أقل 8 بتات من العملية. بينما أصدر فريق Rust مكتبة لتوليد الأرقام العشوائية، إلا أنها ليست جزءًا من المكتبة القياسية، لذا سنضيفها إلى مشروعنا.

في `chip8_core/Cargo.toml`، أضف السطر التالي في مكان ما تحت `[dependencies]`:

```
rand = "^0.7.3"
```

ملاحظة: إذا كنت تخطط لاتباع هذا الدليل حتى النهاية، ستكون هناك تغييرات مستقبلية حول كيفية تضمين هذه المكتبة لدعم متصفح الويب. ومع ذلك، في هذه المرحلة من المشروع، يكفي تحديدها كما هي.

حان الوقت الآن لإضافة دعم توليد الأرقام العشوائية وتنفيذ هذه العملية. في أعلى `lib.rs`، سنحتاج إلى استيراد دالة من مكتبة `rand`:

```
use rand::random;
```

سنستخدم بعد ذلك الدالة `random` عند تنفيذ العملية:

```
match (digit1, digit2, digit3, digit4) {
// -- كود غير متغير محذوف --

// VX = rand() & NN
(0xC, _, _, _) => {
let x = digit2 as usize;
let nn = (op & 0xFF) as u8;
let rng: u8 = random();
self.v_reg[x] = rng & nn;
},

// -- كود غير متغير محذوف --
}
```

لاحظ أن تحديد `rng` كمتغير من النوع `u8` ضروري حتى تعرف الدالة `random()` النوع الذي يجب أن تولده.

DXYN - رسم Sprite

هذه العملية هي على الأرجح الأكثر تعقيدًا، لذا اسمح لي أن أشرح بالتفصيل كيفية عملها. بدلاً من رسم وحدات البكسل الفردية أو المستطيلات على الشاشة في كل مرة، تعرض شاشة Chip-8 *sprites*، وهي صور مخزنة في الذاكرة يتم نسخها إلى الشاشة في إحداثيات محددة (x, y). بالنسبة لهذه العملية، يعطينا الرقمان الثاني والثالث *سجلات V* التي سنستخدمها لجلب إحداثيات (x, y). حتى الآن، الأمور جيدة. Sprites في Chip-8 تكون دائمًا بعرض 8 وحدات بكسل، ولكن يمكن أن يكون ارتفاعها متغيرًا من 1 إلى 16. يتم تحديد ذلك في الرقم الأخير من

العملية. ذكرت سابقًا أن *سجل I* يستخدم بشكل متكرر لتخزين عنوان في الذاكرة، وهذا هو الحال هنا؛ يتم تخزين sprites الخاصة بنا صفًا تلو الآخر *بدءًا* من العنوان المخزن في *I*. لذا إذا طُلب منا رسم sprite بارتفاع 3 وحدات بكسل، فإن بيانات الصف الأول مخزنة في *I*، يليها *I + 1، ثم *I + 2. هذا يفسر سبب كون جميع sprites بعرض 8 وحدات بكسل، حيث يتم تعيين بايت واحد لكل صف، وهو 8 بتات، واحد لكل بكسل، أسود أو أبيض. التفصيل الأخير الذي يجب ملاحظته هو أنه إذا تم تغيير أي بكسل من الأبيض إلى الأسود أو العكس، يتم تعيين *VF* (ويتم مسحه في حالة عدم التغيير). مع وضع هذه الأشياء في الاعتبار، دعنا نبدأ.

```
                match (digit1, digit2, digit3, digit4) {
                    // -- كود غير متغير محذوف --

                    // رسم
                    (0xD, _, _, _) => {
                        // الحصول على إحداثيات (x, y) للـ sprite
                        let x_coord = self.v_reg[digit2 as usize] as u16;
                        let y_coord = self.v_reg[digit3 as usize] as u16;
                        // الرقم الأخير يحدد عدد الصفوف التي يتكون منها الـ sprite
                        let num_rows = digit4;

                        // تتبع إذا تم تغيير أي وحدات بكسل
                        let mut flipped = false;
                        // التكرار على كل صف من الـ sprite
                        for y_line in 0..num_rows {
                            // تحديد عنوان الذاكرة الذي يتم فيه تخزين بيانات الصف
                            let addr = self.i_reg + y_line as u16;
                            let pixels = self.ram[addr as usize];
                            // التكرار على كل عمود في الصف
                            for x_line in 0..8 {
                                // ع لجلب بت البكسل الحالي. يتم التغيير فقط إذا كان البت 1 //
                                if (pixels & (0b1000_0000 >> x_line)) != 0 {
                                    // ى الشاشة، لذا نطبق عملية sprites يجب أن تلتف الـ //
                                    let x = (x_coord + x_line) as usize % SCREEN
                                    let y = (y_coord + y_line) as usize % SCREEN
```

```
                // الحصول على فهرس البكسل لمصفوفة الشاشة أحادية البعد
                let idx = x + SCREEN_WIDTH * y;
                // التحقق مما إذا كنا على وشك تغيير البكسل وتعيين
                flipped |= self.screen[idx];
                self.screen[idx] ^= true;
                }
                }
                }

                // تعيين سجل VF
                if flipped {
                self.v_reg[0xF] = 1;
                } else {
                self.v_reg[0xF] = 0;
                }
                },

                // -- كود غير متغير محذوف --
                }
```

EX9E - تخطي إذا تم الضغط على المفتاح

حان الوقت أخيرًا لإدخال إدخال المستخدم. عند إعداد كائن المحاكاة الخاص بنا، ذكرت أن هناك 16 مفتاحًا ممكنًا مرقمًا من 0 إلى 0xF. تتحقق هذه العملية مما إذا كان المفتاح المخزن في VX مضغوطًا، وإذا كان الأمر كذلك، تتخطى العملية التالية.

```
match (digit1, digit2, digit3, digit4) {
// -- كود غير متغير محذوف --

// تخطي إذا تم الضغط على المفتاح
(0xE, _, 9, 0xE) => {
let x = digit2 as usize;
let vx = self.v_reg[x];
let key = self.keys[vx as usize];
if key {
self.pc += 2;
```

```
            }
        },

        // -- كود غير متغير محذوف --
    }
```

EXA1 - تخطي إذا لم يتم الضغط على المفتاح

نفس العملية السابقة، ولكن هذه المرة يتم تخطي العملية التالية إذا لم يتم الضغط على المفتاح المحدد.

```
    match (digit1, digit2, digit3, digit4) {
        // -- كود غير متغير محذوف --

        // نخطي إذا لم يتم الضغط على المفتاح
        (0xE, _, 0xA, 1) => {
            let x = digit2 as usize;
            let vx = self.v_reg[x];
            let key = self.keys[vx as usize];
            if !key {
                self.pc += 2;
            }
        },

        // -- كود غير متغير محذوف --
    }
```

FX07 - VX = DT

ذكرت استخدام *مؤقت التأخير* عندما كنا نقوم بإعداد بنية المحاكاة. هذا المؤقت ينقص كل إطار حتى يصل إلى الصفر. ومع ذلك، فإن هذه العملية تحدث تلقائيًا، سيكون من المفيد حقًا أن نتمكن من رؤية ما يوجد في *مؤقت التأخير* لأغراض توقيت اللعبة. تقوم هذه العملية بذلك، وتخزن القيمة الحالية في أحد *سجلات* V لاستخدامها.

```
match (digit1, digit2, digit3, digit4) {
// -- كود غير متغير محذوف --

// VX = DT
(0xF, _, 0, 7) => {
let x = digit2 as usize;
self.v_reg[x] = self.dt;
},

// -- كود غير متغير محذوف --
}
```

FX0A - انتظار الضغط على المفتاح

بينما كانت لدينا بالفعل عمليات للتحقق مما إذا كانت المفاتيح مضغوطة أو غير مضغوطة، فإن هذه العملية تفعل شيئًا مختلفًا تمامًا. على عكس تلك العمليات، التي تحقق من حالة المفتاح ثم تستمر، فإن هذه العملية *توقف التنفيذ*، مما يعني أن اللعبة بأكملها ستتوقف وتنتظر طالما كان ذلك ضروريًا حتى يضغط اللاعب على مفتاح. هذا يعني أنها تحتاج إلى التكرار بلا نهاية حتى يصبح شيء ما في مصفوفة `keys` صحيحًا. بمجرد العثور على مفتاح، يتم تخزينه في VX. إذا كان هناك أكثر من مفتاح مضغوط حاليًا، يتم أخذ المفتاح ذي الفهرس الأصغر.

```
match (digit1, digit2, digit3, digit4) {
// -- كود غير متغير محذوف --

// انتظار الضغط على المفتاح
(0xF, _, 0, 0xA) => {
let x = digit2 as usize;
let mut pressed = false;
for i in 0..self.keys.len() {
if self.keys[i] {
self.v_reg[x] = i as u8;
pressed = true;
break;
```

```
                }
                }

                if !pressed {
                // إعادة تنفيذ العملية
                self.pc -= 2;
                }
                },

            // -- كود غير متغير محذوف --
                }
```

قد تنظر إلى هذا التنفيذ وتسأل "لماذا نقوم بإعادة تعيين العملية والمرور عبر تسلسل الجلب بأكمله مرة أخرى، بدلاً من القيام بذلك ببساطة في حلقة؟". ببساطة، بينما نريد أن توقف هذه العملية العمليات المستقبلية من التنفيذ، لا نريد أن توقف أي ضغطات مفاتيح جديدة من التسجيل. من خلال البقاء في حلقة، سنمنع تشغيل كود الضغط على المفتاح، مما يتسبب في عدم انتهاء هذه الحلقة أبدًا. ربما يكون هذا غير فعال، ولكنه أبسط بكثير من بعض أنواع الفحص غير المتزامن.

FX15 - DT = VX

تعمل هذه العملية في الاتجاه المعاكس لعملية *مؤقت التأخير* السابقة. نحتاج إلى طريقة لإعادة تعيين *مؤقت التأخير* إلى قيمة، وتسمح لنا هذه العملية بنسخ قيمة من *سجل* V نختاره.

```
match (digit1, digit2, digit3, digit4) {
// -- كود غير متغير محذوف --

                // DT = VX
(0xF, _, 1, 5) => {
```

```
                let x = digit2 as usize;
                self.dt = self.v_reg[x];
                },

            // -- كود غير متغير محذوف --
            }
```

FX18 - ST = VX

تقريبًا نفس العملية السابقة، ولكن هذه المرة سنقوم بتخزين القيمة من VX في *مؤقت الصوت*.

```
        match (digit1, digit2, digit3, digit4) {
            // -- كود غير متغير محذوف --

                // ST = VX
                (0xF, _, 1, 8) => {
                let x = digit2 as usize;
                self.st = self.v_reg[x];
                },

            // -- كود غير متغير محذوف --
            }
```

FX1E - I += VX

تعمل العملية ANNN على تعيين I إلى القيمة المشفرة 0xNNN، ولكن في بعض الأحيان يكون من المفيد أن نتمكن من زيادة القيمة ببساطة. تأخذ هذه العملية القيمة المخزنة في VX وتضيفها إلى *سجل I*. في حالة التجاوز، يجب أن يعود السجل إلى 0، وهو ما يمكن تحقيقه باستخدام الدالة `wrapping_add` في Rust.

```
        match (digit1, digit2, digit3, digit4) {
            // -- كود غير متغير محذوف --
```

```
                                        // I += VX
                                    (0xF, _, 1, 0xE) => {
                                let x = digit2 as usize;
                            let vx = self.v_reg[x] as u16;
                    self.i_reg = self.i_reg.wrapping_add(vx);
                                                },

                                    // -- كود غير متغير محذوف --
                                                }
```

FX29 - تعيين I إلى عنوان الخط

هذه عملية أخرى صعبة حيث قد لا يكون واضحًا كيفية المضي قدمًا في البداية. إذا كنت تتذكر، قمنا بتخزين مجموعة من بيانات الخط في بداية الذاكرة العشوائية (RAM) عند تهيئة المحاكي. تريد هذه العملية أن نأخذ من VX رقمًا لطباعته على الشاشة (من 0 إلى 0xF)، ونخزن عنوان الذاكرة العشوائية لهذا الـ sprite في *سجل I*. في الواقع، يمكننا تخزين هذه الـ sprites في أي مكان نريده، طالما أننا متسقون ونشير إليها بشكل صحيح هنا. ومع ذلك، قمنا بتخزينها في موقع مناسب جدًا، في بداية الذاكرة العشوائية. دعني أوضح لك ما أعنيه عن طريق طباعة بعض الـ sprites وعناوين الذاكرة العشوائية الخاصة بها.

الحرف عنوان الذاكرة العشوائية

0	0
5	1
10	2
15	3
…	…
70	E (14)
75	F (15)

ستلاحظ أنه نظرًا لأن جميع الـ sprites الخاصة بالخط تأخذ خمسة بايتات لكل منها، فإن عنوان الذاكرة العشوائية الخاص بها هو ببساطة قيمتها مضروبة في 5. إذا قمنا بتخزين الخطوط في عنوان ذاكرة عشوائية مختلف، فلا يزال بإمكاننا اتباع هذه القاعدة، ولكن يجب علينا تطبيق إزاحة على مكان بدء الكتلة.

```
match (digit1, digit2, digit3, digit4) {
// -- كود غير متغير محذوف --

                    // I = FONT
                (0xF, _, 2, 9) => {
            let x = digit2 as usize;
        let c = self.v_reg[x] as u16;
            self.i_reg = c * 5;
                                },

// -- كود غير متغير محذوف --
                                }
```

VX لـ FX33 - I = BCD

معظم عمليات Chip-8 واضحة إلى حد ما، ويمكن تنفيذها بسهولة بمجرد سماع وصف غامض. ومع ذلك، هناك بعض العمليات الصعبة، مثل الرسم على الشاشة وهذه العملية، التي تقوم بتخزين الرقم العشري المشفر ثنائيًا (BCD) لرقم مخزن في VX في *سجل I*. أشجعك على القراءة عن BCD إذا كنت غير معتاد عليها، ولكن هنا شرح سريع: في هذا البرنامج التعليمي، استخدمنا النظام الست عشري (hexadecimal) كثيرًا، والذي يعمل عن طريق تحويل أرقامنا العشرية العادية إلى نظام الأساس 16، وهو ما تفهمه أجهزة الكمبيوتر بسهولة أكبر. على سبيل المثال، الرقم العشري 100 سيصبح 0x64. هذا جيد لأجهزة

الكمبيوتر، ولكنه ليس سهل الوصول للبشر، وبالتأكيد ليس للجمهور العام الذي سيلعب ألعابك. الغرض الرئيسي من BCD هو تحويل الرقم الست عشري مرة أخرى إلى رقم عشري شبه عشري لطباعته للمستخدم، مثل النقاط أو النتائج العالية. لذا بينما قد يخزن Chip-8 الرقم 0x64 داخليًا، فإن جلب BCD الخاص به سيعطينا `0x1, 0x0, 0x0`، والتي يمكننا طباعتها على الشاشة كـ "100".

ستلاحظ أننا انتقلنا من بايت واحد إلى ثلاثة بايتات لتخزين جميع الأرقام الثلاثة لرقمنا، ولهذا السبب سنقوم بتخزين BCD في الذاكرة العشوائية، بدءًا من العنوان الموجود حاليًا في *سجل I* والمضي قدمًا. نظرًا لأن VX تخزن أرقامًا من 8 بت، والتي تتراوح من 0 إلى 255، فسنحصل دائمًا على ثلاثة بايتات، حتى إذا كانت بعضها صفرًا.

```
                match (digit1, digit2, digit3, digit4) {
                                // -- كود غير متغير محذوف --

                                        // BCD
                                    (0xF, _, 3, 3) => {
                                let x = digit2 as usize;
                            let vx = self.v_reg[x] as f32;

                // جلب رقم المئات عن طريق القسمة على 100 وإزالة الكسور
                                let hundreds = (vx / 100.0).floor() as u8;
// جلب رقم العشرات عن طريق القسمة على 10، وإزالة رقم الآحاد والكسور
                                let tens = ((vx / 10.0) % 10.0).floor() as u8;
                                // جلب رقم الآحاد عن طريق إزالة المئات والعشرات
                                let ones = (vx % 10.0) as u8;

                                self.ram[self.i_reg as usize] = hundreds;
                            self.ram[(self.i_reg + 1) as usize] = tens;
                            self.ram[(self.i_reg + 2) as usize] = ones;
                                                                    },
```

```
// -- كود غير متغير محذوف --
}
```

بالنسبة لهذا التنفيذ، قمت بتحويل قيمة VX أولاً إلى `float`، حتى أتمكن من استخدام عمليات القسمة والـ modulo للحصول على كل رقم عشري. هذا ليس أسرع تنفيذ ولا الأقصر. ومع ذلك، فهو أحد أسهل الطرق للفهم. أنا متأكد من أن هناك بعض القراء المطلعين على النظام الثنائي الذين يشعرون بالاشمئزاز من أنني فعلت ذلك بهذه الطريقة، ولكن هذا الحل ليس لهم. هذا الحل موجه للقراء الذين لم يسبق لهم رؤية BCD من قبل، حيث أن فقدان بعض السرعة من أجل فهم أفضل هو مقايضة جيدة. ومع ذلك، بمجرد تنفيذ هذا، أشجع الجميع على البحث عن خوارزميات BCD أكثر كفاءة لإضافة بعض التحسينات السهلة إلى الكود.

FX55 - تخزين V0 - VX في I

نحن على وشك الانتهاء! تقوم هاتان العمليتان الأخيرتان بملء *سجلات* *V* من V0 إلى VX المحدد (بما في ذلك) بنفس النطاق من القيم من الذاكرة العشوائية، بدءًا من العنوان الموجود في *سجل* *I*. تقوم العملية الأولى بتخزين القيم في الذاكرة العشوائية، بينما تقوم العملية التالية بتحميلها في الاتجاه المعاكس.

```
match (digit1, digit2, digit3, digit4) {
// -- كود غير متغير محذوف --

// تخزين V0 - VX
(0xF, _, 5, 5) => {
let x = digit2 as usize;
let i = self.i_reg as usize;
for idx in 0..=x {
```

```
                self.ram[i + idx] = self.v_reg[idx];
                }
            },

            // -- كود غير متغير محذوف --
            }
```

FX65 - تحميل I إلى V0 - VX

```
        match (digit1, digit2, digit3, digit4) {
            // -- كود غير متغير محذوف --

            // تحميل V0 - VX
            (0xF, _, 6, 5) => {
                let x = digit2 as usize;
                let i = self.i_reg as usize;
                for idx in 0..=x {
                    self.v_reg[idx] = self.ram[i + idx];
                }
            },

            // -- كود غير متغير محذوف --
            }
```

الأفكار النهائية

هذا كل شيء! مع هذا، أصبح لدينا الآن وحدة معالجة مركزية Chip-8 مكتملة التنفيذ. ربما لاحظت أن العديد من قيم الأوامر المشفرة (opcodes) لم يتم تغطيتها، خاصة في نطاقات 0x0000 و0xE000 و0xF000. هذا أمر طبيعي. هذه الأوامر المشفرة غير محددة في التصميم الأصلي، وبالتالي إذا حاولت أي لعبة استخدامها، فسيؤدي ذلك إلى حدوث خطأ في وقت التشغيل. إذا كنت لا تزال فضوليًا بعد الانتهاء من هذا المحاكي، فهناك عدد من امتدادات Chip-8 التي

تملأ بعض هذه الفجوات لإضافة وظائف إضافية، ولكنها لن يتم تغطيتها في هذا الدليل.

كتابة الواجهة الأمامية لسطح المكتب

في هذا القسم، سنقوم أخيرًا بتوصيل جميع القطع وجعل المحاكي الخاص بنا يقوم بتحميل وتشغيل لعبة. في هذه المرحلة، أصبحت نواة المحاكي قادرة على تحليل ومعالجة أوامر اللعبة، وتحديث الشاشة والذاكرة والسجلات حسب الحاجة. بعد ذلك، سنحتاج إلى إضافة بعض الوظائف العامة لتوفير بعض الوظائف للواجهة الأمامية، مثل تحميل لعبة، قبول إدخال المستخدم، ومشاركة مخزن الشاشة ليتم عرضه.

تعريض النواة للواجهة الأمامية

لم ننتهي بعد من `chip8_core`. نحتاج إلى إضافة بعض الوظائف العامة إلى بنية `Emu` لإعطاء الوصول إلى بعض العناصر.

في `chip8_core/src/lib.rs`، أضف الطريقة التالية إلى بنية `Emu`:

```
impl Emu {
// -- كود غير متغير محذوف --

pub fn get_display(&self) -> &[bool] {
&self.screen
}

// -- كود غير متغير محذوف --
}
```

هذا ببساطة يمرر مؤشرًا إلى مصفوفة مخزن الشاشة إلى الواجهة الأمامية، حيث يمكن استخدامها لعرض الشاشة.

بعد ذلك، سنحتاج إلى التعامل مع ضغطات المفاتيح. لدينا بالفعل مصفوفة keys، ولكنها لم تتم كتابتها أبدًا. ستقوم الواجهة الأمامية بالتعامل مع قراءة ضغطات المفاتيح، ولكننا سنحتاج إلى تعريض وظيفة تسمح لها بالتفاعل وتعيين العناصر في مصفوفة keys.

```
impl Emu {
// -- كود غير متغير محذوف --

pub fn keypress(&mut self, idx: usize, pressed: bool) {
self.keys[idx] = pressed;
}

// -- كود غير متغير محذوف --
}
```

هذه الوظيفة بسيطة جدًا. تأخذ الفهرس الخاص بالمفتاح الذي تم الضغط عليه وتعيين القيمة. كان بإمكاننا تقسيم هذه الوظيفة إلى press_key و release_key، ولكن هذا بسيط بما يكفي بحيث يظل الهدف واضحًا. تذكر أن idx يجب أن يكون أقل من 16 وإلا ستفشل البرنامج. يمكنك إضافة هذا القيد هنا، ولكن بدلاً من ذلك سنتعامل معه في الواجهة الأمامية ونفترض أنه تم التحقق منه بشكل صحيح في الخلفية، بدلاً من التحقق منه مرتين.

أخيرًا، نحتاج إلى طريقة لتحميل كود اللعبة من ملف إلى ذاكرتنا حتى يمكن تنفيذه. سنتعمق في هذا أكثر عندما نبدأ في القراءة من ملف في الواجهة

الأمامية، ولكن في الوقت الحالي نحتاج إلى أخذ قائمة من البايتات ونسخها إلى ذاكرتنا.

```
impl Emu {
// -- كود غير متغير محذوف --

    pub fn load(&mut self, data: &[u8]) {
        let start = START_ADDR as usize;
let end = (START_ADDR as usize) + data.len();
  self.ram[start..end].copy_from_slice(data);
    }

// -- كود غير متغير محذوف --
}
```

تقوم هذه الوظيفة بنسخ جميع القيم من شريحة الإدخال `data` إلى الذاكرة بدءًا من 0x200. تذكر أن أول 512 بايت من الذاكرة لا تحتوي على بيانات اللعبة، وهي فارغة باستثناء بيانات الـ sprites التي نقوم بتخزينها هناك.

إعداد الواجهة الأمامية

أخيرًا، لنقم بإعداد الواجهة الأمامية للمحاكي حتى نتمكن من اختبار الأشياء ونأمل أن نلعب بعض الألعاب! إنشاء واجهة مستخدم رسومية متقدمة هو خارج نطاق هذا الدليل، سنقوم ببساطة ببدء المحاكي واختيار اللعبة التي نريد لعبها عبر وسيطة سطر الأوامر. لنقم بإعداد ذلك الآن.

في `desktop/src/main.rs`، سنحتاج إلى قراءة وسيطات سطر الأوامر لتلقي مسار ملف لعبة ROM الخاص بنا. كان بإمكاننا إنشاء عدة أعلام لإعدادات

إضافية، ولكننا سنبقيها بسيطة ونقول أننا سنطلب وسيطة واحدة بالضبط - مسار اللعبة. أي عدد آخر وسنخرج بخطأ.

```
use std::env;

fn main() {
let args: Vec<_> = env::args().collect();
if args.len() != 2 {
println!("Usage: cargo run path/to/game");
return;
}
}
```

هذا يأخذ جميع معلمات سطر الأوامر الممررة إلى متجه، وإذا لم يكن هناك اثنان (اسم البرنامج دائمًا مخزن في `args[0]`)، فإننا نطبع الإدخال الصحيح ونخرج. المسار الذي أدخله المستخدم موجود الآن في `args[1]`. سنحتاج إلى التأكد من أن هذا الملف صالح بمجرد محاولة فتحه، ولكن أولاً، لدينا بعض الأشياء الأخرى لإعدادها.

إنشاء نافذة

لمشروع المحاكاة هذا، سنستخدم مكتبة SDL لإنشاء نافذة اللعبة والرسم عليها. SDL هي مكتبة رسم ممتازة مع دعم جيد لضغطات المفاتيح والرسم. هناك كمية صغيرة من الكود التمهيدي التي سنحتاجها لإعدادها، ولكن بمجرد الانتهاء من ذلك يمكننا بدء المحاكاة.

أولاً، سنحتاج إلى تضمينها في مشروعنا. افتح `desktop/Cargo.toml` وأضف `sdl2` إلى التبعيات:

```
[dependencies]
chip8_core = { path = "../chip8_core" }
sdl2 = "^0.34.3
```

الآن في `desktop/src/main.rs`، يمكننا البدء في تجميع كل شيء معًا. سنحتاج إلى الوظائف العامة التي حددناها في نواتنا، لذا دعنا نخبر Rust أننا بحاجة إليها باستخدام `use chip8_core::*;`.

```
use chip8_core::*;
use std::env;

fn main() {
let args: Vec<_> = env::args().collect();
if args.len() != 2 {
println!("Usage: cargo run path/to/game");
return;
}
}
```

داخل `chip8_core`، قمنا بإنشاء ثوابت عامة لتخزين حجم الشاشة، والتي نقوم الآن باستيرادها. ومع ذلك، نافذة لعبة بحجم 64x32 صغيرة جدًا على شاشات اليوم، لذا دعنا نضاعف حجمها قليلاً. بعد تجربة بعض الأرقام، وجدت أن مقياس 15x يعمل بشكل جيد على شاشتي، ولكن يمكنك تعديل هذا إذا كنت تفضل شيئًا آخر.

```
const SCALE: u32 = 15;
const WINDOW_WIDTH: u32 = (SCREEN_WIDTH as u32) * SCALE;
const WINDOW_HEIGHT: u32 = (SCREEN_HEIGHT as u32) * SCALE;
```

تذكر أن `SCREEN_WIDTH` و `SCREEN_HEIGHT` كانتا ثوابت عامة حددناهما في الخلفية ويتم تضمينهما الآن في هذه الحزمة عبر عبارة `use chip8_core::*`. SDL يتطلب أن تكون أحجام الشاشة من النوع `u32` بدلاً من `usize` لذا سنقوم بتحويلها هنا.

حان الوقت لإنشاء نافذة SDL الخاصة بنا! الكود التالي ببساطة ينشئ سياق SDL جديد، ثم يقوم بإنشاء النافذة نفسها واللوحة التي سنرسم عليها.

```
                                        fn main() {
                                    // -- كود غير متغير محذوف --

                                            // إعداد SDL
                                let sdl_context = sdl2::init().unwrap();
                            let video_subsystem = sdl_context.video().unwrap();
                                        let window = video_subsystem
                    .window("Chip-8 Emulator", WINDOW_WIDTH, WINDOW_HEIGHT)
                                                .position_centered()
                                                        .opengl()
                                                         .build()
                                                        .unwrap();

            let mut canvas = window.into_canvas().present_vsync().build(
                                                        canvas.clear();
                                                    canvas.present();
                                                        }
```

سنقوم بتهيئة SDL ونطلب منها إنشاء نافذة جديدة بحجمنا المضاعف. سنقوم أيضًا بوضعها في منتصف شاشة المستخدم. بعد ذلك، سنحصل على كائن اللوحة الذي سنرسم عليه، مع تفعيل VSYNC. ثم نقوم بمسحها وعرضها للمستخدم.

إذا حاولت تشغيلها الآن (أعطها اسم ملف وهمي للاختبار، مثل `cargo run test`)، ستظهر نافذة لفترة وجيزة قبل أن تغلق. هذا لأن نافذة SDL يتم إنشاؤها لفترة وجيزة، ولكن بعد ذلك ينتهي البرنامج وتغلق النافذة. سنحتاج إلى إنشاء حلقة اللعبة الرئيسية حتى لا ينتهي البرنامج فورًا، وبينما نحن في ذلك، دعنا نضيف بعض التعامل مع الخروج من النافذة إذا حاولنا إغلاقها (وإلا سيتعين عليك إجبار البرنامج على الإغلاق من مدير المهام).

يستخدم SDL شيئًا يسمى *مضخة الأحداث* للتحقق من الأحداث في كل حلقة. عن طريق التحقق من هذا، يمكننا جعل أشياء مختلفة تحدث لأحداث معينة، مثل محاولة إغلاق النافذة أو الضغط على مفتاح. في الوقت الحالي، سنقوم فقط بجعل البرنامج يخرج من حلقة اللعبة الرئيسية إذا احتاج إلى إغلاق النافذة.

سنحتاج إلى إخبار Rust أننا نرغب في استخدام `Event` من SDL:

```
use sdl2::event::Event;
```

وتعديل دالة `main` لإضافة حلقة اللعبة الأساسية:

```
fn main() {
// -- كود غير متغير محذوف --

// إعداد SDL
    let sdl_context = sdl2::init().unwrap();
    let video_subsystem = sdl_context.video().unwrap();
    let window = video_subsystem
        .window("Chip-8 Emulator", WINDOW_WIDTH, WINDOW_HEIGHT)
        .position_centered()
        .opengl()
        .build()
        .unwrap();
```

```
    let mut canvas = window.into_canvas().present_vsync().build(
                                                    canvas.clear();
                                                canvas.present();

        let mut event_pump = sdl_context.event_pump().unwrap();

                                                'gameloop: loop {
                                            for evt in event_pump.poll_iter() {
                                                        match evt {
                                                    Event::Quit{..} => {
                                                        break 'gameloop;
                                                            },
                                                    _ => ()
                                                    }
                                                    }
                                                    }
                                                    }
```

هذه الإضافة تقوم بإعداد حلقة اللعبة الرئيسية، والتي تتحقق مما إذا كانت أي أحداث قد تم تشغيلها. إذا تم اكتشاف حدث `Quit` (عن طريق محاولة إغلاق النافذة)، فإن البرنامج يخرج من الحلقة، مما يؤدي إلى إنهائه. إذا حاولت تشغيلها مرة أخرى عبر `cargo run test`، ستظهر نافذة سوداء جديدة بعنوان 'Chip-8 Emulator'. يجب أن تغلق النافذة بنجاح دون مشاكل.

إنها تعمل! بعد ذلك، سنقوم بتهيئة نواة المحاكي `chip8_core`، وفتح وتحميل ملف اللعبة.

تحميل ملف

الواجهة الأمامية لدينا يمكنها الآن إنشاء نافذة محاكاة جديدة، لذا حان الوقت لبدء تشغيل الخلفية أيضًا. ستكون خطوتنا التالية هي قراءة ملف لعبة فعليًا، وتمرير

بياناته إلى الخلفية ليتم تخزينها في الذاكرة وتنفيذها. أولاً، نحتاج إلى وجود كائن خلفي لتمرير الأشياء إليه. لا يزال في `frontend/src/main.rs`، لنقم بإنشاء كائن المحاكاة الخاص بنا. تم بالفعل تضمين كائن `Emu` مع جميع العناصر العامة من `chip8_core`، لذا يمكننا التهيئة بحرية.

```
fn main() {
// -- كود غير متغير محذوف --

let mut chip8 = Emu::new();

'gameloop: loop {
// -- كود غير متغير محذوف --
}
}
```

إنشاء كائن `Emu` يجب أن يتم في مكان ما قبل حلقة اللعبة الرئيسية، حيث سيتم فيها رسم المحاكاة والتعامل مع ضغطات المفاتيح. تذكر أن مسار اللعبة يتم تمريره من قبل المستخدم، لذا سنحتاج أيضًا إلى التأكد من أن هذا الملف موجود بالفعل قبل محاولة قراءته. سنحتاج أولاً إلى `use` بعض العناصر من المكتبة القياسية في `main.rs` لفتح ملف.

```
use std::fs::File;
use std::io::Read;
```

هذا واضح جدًا. بعد ذلك، سنقوم بفتح الملف المعطى لنا كمعامل سطر أوامر، وقراءته في مخزن مؤقت، ثم تمرير تلك البيانات إلى خلفية المحاكي.

```
fn main() {
// -- كود غير متغير محذوف --
let mut chip8 = Emu::new();
```

```
    let mut rom = File::open(&args[1]).expect("Unable to open fi
                                        let mut buffer = Vec::new();

                                    rom.read_to_end(&mut buffer).unwrap();
                                                chip8.load(&buffer);
                                            // -- كود غير متغير محذوف --
                                                                    }
```

هناك بعض الأشياء التي يجب ملاحظتها هنا. في حالة عدم تمكن Rust من فتح الملف من المسار الذي أعطانا إياه المستخدم (على الأرجح لأنه غير موجود)، فإن شرط `expect` سيفشل وسيخرج البرنامج مع هذه الرسالة. ثانيًا، كان بإمكاننا إعطاء مسار الملف إلى الخلفية وتحميل البيانات هناك، ولكن قراءة الملف هي سلوك أكثر ملاءمة للواجهة الأمامية وهو أفضل هنا. والأهم من ذلك، خطتنا النهائية هي جعل هذا المحاكي يعمل في متصفح الويب مع تغييرات قليلة أو معدومة في الخلفية. كيفية قراءة المتصفح لملف تختلف كثيرًا عن كيفية قراءة نظام الملفات الخاص بك، لذا سنسمح للواجهات الأمامية بالتعامل مع القراءة، وتمرير البيانات بمجرد حصولنا عليها.

تشغيل المحاكي والرسم على الشاشة

تم تحميل اللعبة في الذاكرة وحلقتنا الرئيسية تعمل. الآن نحتاج إلى إخبار الخلفية ببدء معالجة تعليماتها، والرسم فعليًا على الشاشة. إذا كنت تتذكر، فإن المحاكي يعمل من خلال دورة ساعة كلما تم استدعاء دالة `tick`، لذا دعنا نضيف ذلك إلى حلقتنا.

```
                                                fn main() {
                                            // -- كود غير متغير محذوف --
```

```
                'gameloop: loop {
            for event in event_pump.poll_iter() {
                // -- كود غير متغير محذوف --
                }

                chip8.tick();
                }
                }
```

الآن في كل مرة تدور فيها الحلقة، سيتقدم المحاكي من خلال تعليمة أخرى. قد يبدو هذا سهلاً للغاية، ولكننا قمنا بإعداده بحيث تقوم `tick` بتحريك جميع أجزاء الخلفية، بما في ذلك تعديل مخزن الشاشة. دعنا نضيف دالة ستجلب بيانات الشاشة من الخلفية وتحديث نافذة SDL الخاصة بنا. أولاً، نحتاج إلى `use` بعض العناصر الإضافية من SDL:

```
use sdl2::pixels::Color;
use sdl2::rect::Rect;
use sdl2::render::Canvas;
use sdl2::video::Window;
```

بعد ذلك، الدالة، التي ستأخذ مرجعًا إلى كائن `Emu` الخاص بنا، بالإضافة إلى مرجع قابل للتعديل إلى لوحة SDL الخاصة بنا. يتطلب رسم الشاشة بضع خطوات. أولاً، نقوم بمسح اللوحة لمحو الإطار السابق. بعد ذلك، نكرر من خلال مخزن الشاشة، ونرسم مستطيلًا أبيض في أي وقت تكون القيمة المعطاة صحيحة. نظرًا لأن Chip-8 تدعم الأسود والأبيض فقط؛ إذا قمنا بمسح الشاشة باللون الأسود، فإننا نحتاج فقط إلى الاهتمام برسم المربعات البيضاء.

```
fn draw_screen(emu: &Emu, canvas: &mut Canvas<Window>) {
    // مسح اللوحة باللون الأسود
    canvas.set_draw_color(Color::RGB(0, 0, 0));
```

```
                canvas.clear();

                let screen_buf = emu.get_display();
// لون الرسم إلى الأبيض، والتكرار خلال كل نقطة لمعرفة ما إذا كان يجب رسمها
                canvas.set_draw_color(Color::RGB(255, 255, 255));
                for (i, pixel) in screen_buf.iter().enumerate() {
                    if *pixel {
// تحويل الفهرس من المصفوفة أحادية البعد إلى موضع ثنائي الأبعاد (
                        let x = (i % SCREEN_WIDTH) as u32;
                        let y = (i / SCREEN_WIDTH) as u32;

// رسم مستطيل في الموضع (x, y)، مع تكبيره حسب قيمة `SCAL
                        let rect = Rect::new((x * SCALE) as i32, (y * SCALE)
                        canvas.fill_rect(rect).unwrap();
                    }
                }
                canvas.present();
            }
```

لتلخيص هذه الوظيفة، نحصل على مصفوفة الشاشة أحادية البعد ونكرر خلالها.

إذا وجدنا بكسلًا أبيض (قيمة `true`)، فإننا نحسب الموضع ثنائي الأبعاد (x, y) للشاشة ونرسم مستطيلًا هناك، مع تكبيره حسب عامل `SCALE`.

سنستدعي هذه الوظيفة في الحلقة الرئيسية، مباشرة بعد استدعاء `tick`.

```
fn main() {
    // -- الكود غير المتغير محذوف --

    'gameloop: loop {
        for event in event_pump.poll_iter() {
            // -- الكود غير المتغير محذوف --
        }

        chip8.tick();
        draw_screen(&chip8, &mut canvas);
    }
}
```

قد يتساءل بعضكم عن هذا. يجب تحديث الشاشة بمعدل 60 إطارًا في الثانية، أو 60 هرتز. بالتأكيد يجب أن تكون المحاكاة أسرع من ذلك؟ نعم، ولكن دعونا نضع هذا التفكير جانبًا الآن. سنبدأ بالتأكد من أن الأمر يعمل بشكل أساسي قبل أن نصلح التوقيتات.

إذا كان لديك لعبة Chip-8، يمكنك تجربة تشغيل المحاكي الخاص بك مع لعبة فعلية عبر:

```
$ cargo run path/to/game
```

إذا سار كل شيء بشكل جيد، يجب أن تظهر النافذة وتبدأ اللعبة في التقديم واللعب! يجب أن تشعر بالإنجاز لوصولك إلى هذا الحد مع محاكيك الخاص.

كما ذكرت سابقًا، يجب أن تكون سرعة `tick` للمحاكاة أسرع من معدل تحديث اللوحة. إذا شاهدت لعبتك تعمل، قد تشعر بأنها بطيئة بعض الشيء. حاليًا، ننفذ تعليمة واحدة، ثم نرسم على الشاشة، ثم نكرر. كما تعلمون، يتطلب الأمر عدة تعليمات لإجراء أي تغييرات ذات معنى على الشاشة. للتغلب على هذا، سنسمح للمحاكي بالتنفيذ عدة مرات قبل إعادة الرسم.

الآن، هذا هو المكان الذي تصبح فيه الأمور تجريبية بعض الشيء. مواصفات Chip-8 لا تقول أي شيء عن مدى سرعة النظام. حتى تركها كما هي الآن لتشغيلها بسرعة 60 هرتز هو حل صالح (وأنت مرحب بفعل ذلك). سنسمح ببساطة لوظيفة `tick` بالتكرار عدة مرات قبل الانتقال إلى رسم الشاشة. شخصيًا،

أجد (وغيري من المحاكيات التي نظرت إليها) أن 10 تكرارات لكل إطار هو نقطة مثالية.

```
const TICKS_PER_FRAME: usize = 10;
// -- الكود غير المتغير محذوف --

fn main() {
// -- الكود غير المتغير محذوف --

'gameloop: loop {
for event in event_pump.poll_iter() {
// -- الكود غير المتغير محذوف --
}

for _ in 0..TICKS_PER_FRAME {
chip8.tick();
}
draw_screen(&chip8, &mut canvas);
}
}
```

قد يشعر بعضكم أن هذا غير دقيق بعض الشيء (وأنا أتفق معكم إلى حد ما). ومع ذلك، هذه أيضًا هي الطريقة التي تعمل بها الأنظمة الأكثر "تطورًا"، مع استثناء أن تلك المعالجات عادةً ما يكون لديها طريقة لإعلام الشاشة بأنها جاهزة لإعادة الرسم. نظرًا لأن Chip-8 لا تملك مثل هذه الآلية ولا أي سرعة ساعة محددة، فإن هذه طريقة أسهل لإنجاز هذه المهمة.

إذا قمت بالتشغيل مرة أخرى، قد تلاحظ أن الأمر لا يصل بعيدًا قبل التوقف. هذا على الأرجح بسبب أننا لم نقم بتحديث المؤقتين لدينا، لذا فإن المحاكي ليس لديه مفهوم عن مقدار الوقت الذي مر لألعابه. لقد ذكرت سابقًا أن المؤقتين

يعملان مرة واحدة لكل إطار، بدلًا من سرعة الساعة، لذا يمكننا تعديل المؤقتين في نفس النقطة التي نعدل فيها الشاشة.

```
fn main() {
// -- الكود غير المتغير محذوف --

'gameloop: loop {
for event in event_pump.poll_iter() {
// -- الكود غير المتغير محذوف --
}

for _ in 0..TICKS_PER_FRAME {
chip8.tick();
}
chip8.tick_timers();
draw_screen(&chip8, &mut canvas);
}
}
```

لا يزال هناك بعض الأشياء المتبقية للتنفيذ (على سبيل المثال، لا يمكنك التحكم في لعبتك) ولكنها بداية رائعة، ونحن في المراحل النهائية الآن!

إضافة إدخال المستخدم

يمكننا أخيرًا تقديم لعبة Chip-8 إلى الشاشة، ولكننا لا نستطيع التقدم كثيرًا في اللعب لأننا لا نملك طريقة للتحكم بها. لحسن الحظ، يدعم SDL قراءة المدخلات من لوحة المفاتيح والتي يمكننا ترجمتها وإرسالها إلى محاكاتنا الخلفية.

كمراجعة، يدعم نظام Chip-8 16 مفتاحًا مختلفًا. هذه المفاتيح عادةً ما تكون منظمة في شبكة 4x4، مع المفاتيح من 0 إلى 9 مرتبة مثل الهاتف والمفاتيح من A إلى F محيطة بها. بينما يمكنك تنظيم المفاتيح بأي تكوين تريده، افترض بعض

مطوري الألعاب أنها في نمط الشبكة عند اختيار مدخلات ألعابهم، مما يعني أنه يمكن أن يكون من الصعب لعب بعض الألعاب بطريقة أخرى. بالنسبة لمحاكاتنا، سنستخدم المفاتيح اليسرى من لوحة المفاتيح QWERTY كمدخلاتنا، كما هو موضح أدناه.

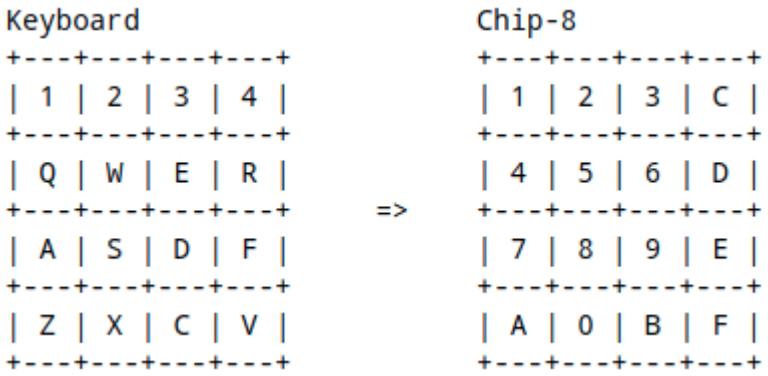
لنقم بإنشاء وظيفة لتحويل نوع المفتاح في SDL إلى القيم التي سنرسلها إلى المحاكي. سنحتاج إلى إدخال دعم لوحة المفاتيح في `main.rs` عبر:

```
use sdl2::keyboard::Keycode;
```

الآن، سننشئ وظيفة جديدة تأخذ `Keycode` وتخرج قيمة `u8` اختيارية. هناك فقط 16 مفتاحًا صالحًا، لذا سنلف قيمة الإخراج الصالحة في `Some`، ونعيد `None` إذا ضغط المستخدم على مفتاح غير Chip-8. هذه الوظيفة هي مجرد مطابقة لجميع المفاتيح الصالحة كما هو موضح في الصورة أعلاه.

```
fn key2btn(key: Keycode) -> Option<usize> {
                                match key {
                    Keycode::Num1 =>    Some(0x1),
                    Keycode::Num2 =>    Some(0x2),
                    Keycode::Num3 =>    Some(0x3),
                    Keycode::Num4 =>    Some(0xC),
                    Keycode::Q =>       Some(0x4),
                    Keycode::W =>       Some(0x5),
                    Keycode::E =>       Some(0x6),
                    Keycode::R =>       Some(0xD),
                    Keycode::A =>       Some(0x7),
                    Keycode::S =>       Some(0x8),
                    Keycode::D =>       Some(0x9),
                    Keycode::F =>       Some(0xE),
                    Keycode::Z =>       Some(0xA),
                    Keycode::X =>       Some(0x0),
                    Keycode::C =>       Some(0xB),
```

```
            Keycode::V =>         Some(0xF),
                _ =>                  None,
                }
                }
```



تخطيط لوحة المفاتيح إلى مفاتيح Chip-8

بعد ذلك، سنضيف حدثين إضافيين إلى حلقة الأحداث الرئيسية، أحدهما لـ `KeyDown` والآخر لـ `KeyUp`. سيتم التحقق من كل حدث إذا كان المفتاح المضغوط يعطي قيمة `Some` من وظيفة `key2btn`، وإذا كان الأمر كذلك، يتم تمريرها إلى المحاكي عبر وظيفة `keypress` العامة التي قمنا بتعريفها سابقًا. الاختلاف الوحيد بين الاثنين هو ما إذا كان يتم تعيينها أو مسحها.

```
                fn main() {
    // -- الكود غير المتغير محذوف --
                'gameloop: loop {
    for evt in event_pump.poll_iter() {
                match evt {
            Event::Quit{..} => {
                break 'gameloop;
                },
    Event::KeyDown{keycode: Some(key), ..} => {
            if let Some(k) = key2btn(key) {
                chip8.keypress(k, true);
                }
                },
```

```
                Event::KeyUp{keycode: Some(key), ..} => {
                    if let Some(k) = key2btn(key) {
                        chip8.keypress(k, false);
                    }
                },
                _ => ()
            }

            for _ in 0..TICKS_PER_FRAME {
                chip8.tick();
            }
            chip8.tick_timers();
            draw_screen(&chip8, &mut canvas);
        }
        // -- الكود غير المتغير محذوف --
    }
```

عبارة `if let` تكون محققة فقط إذا كانت القيمة على اليمين تطابق تلك على اليسار، أي أن `key2btn(key)` تعيد قيمة `Some`. القيمة التي تم فكها يتم تخزينها بعد ذلك في `k`.

لنضيف أيضًا قدرة شائعة في المحاكيات - إغلاق البرنامج بالضغط على Escape.

سنضيف ذلك بجانب حدث `Quit`.

```
fn main() {
    // -- الكود غير المتغير محذوف --
    'gameloop: loop {
        for evt in event_pump.poll_iter() {
            match evt {
                Event::Quit{..} | Event::KeyDown{keycode: Some(K
                    break 'gameloop;
                },
                Event::KeyDown{keycode: Some(key), ..} => {
                    if let Some(k) = key2btn(key) {
                        chip8.keypress(k, true);
                    }
                },
```

```
                Event::KeyUp{keycode: Some(key), ..} => {
                    if let Some(k) = key2btn(key) {
                        chip8.keypress(k, false);
                    }
                },
                _ => ()
            }
        }

        for _ in 0..TICKS_PER_FRAME {
            chip8.tick();
        }
        chip8.tick_timers();
        draw_screen(&chip8, &mut canvas);
    }
    // -- الكود غير المتغير محذوف --
}
```

على عكس أحداث المفاتيح الأخرى، حيث نتحقق من المتغير key الذي تم العثور عليه، نريد استخدام مفتاح Escape للإغلاق. إذا كنت لا تريد هذه القدرة في محاكاتك، أو تريد بعض الوظائف الأخرى للضغط على المفاتيح، فأنت مرحب بذلك.

هذا كل شيء! الواجهة الأمامية لمحاكي Chip-8 لدينا على سطح المكتب مكتملة الآن. يمكننا تحديد لعبة عبر معلمة سطر الأوامر، وتحميلها وتنفيذها، وعرض الإخراج على الشاشة، والتعامل مع إدخال المستخدم.

آمل أن تكون قد تمكنت من فهم كيفية عمل المحاكاة. نظام Chip-8 هو نظام بسيط إلى حد ما، ولكن التقنيات التي تمت مناقشتها هنا تشكل الأساس لكيفية عمل جميع المحاكيات.

ومع ذلك، هذا الدليل لم ينته بعد! في القسم التالي سأناقش كيفية بناء محاكاتنا باستخدام WebAssembly وجعله يعمل في متصفح الويب.

مقدمة إلى WebAssembly

سنتحدث في هذا القسم عن كيفية تحويل المحاكي الذي أنشأناه لتشغيله في متصفح الويب باستخدام تقنية جديدة نسبيًا تسمى *WebAssembly*. أشجعك على قراءة المزيد عن WebAssembly. إنها صيغة لتحويل البرامج إلى ملف تنفيذي ثنائي، مشابه في النطاق لملف .exe، ولكنها مخصصة للتشغيل داخل متصفح الويب. يتم دعمها من قبل جميع المتصفحات الرئيسية، وهي معيار متعدد الشركات يتم تطويره بينها. هذا يعني أنه بدلًا من الاضطرار إلى كتابة كود الويب باستخدام JavaScript أو لغات أخرى مخصصة للويب، يمكنك كتابته بأي لغة تدعم تحويل ملفات .wasm ولا يزال بإمكانك التشغيل في المتصفح. في وقت كتابة هذا الدليل، تعد C وC++ وRust اللغات الرئيسية التي تدعمها، ولحسن الحظ بالنسبة لنا.

الإعداد

بينما يمكننا التحويل يدويًا إلى أهداف WebAssembly في Rust، تم تطوير مجموعة من الأدوات المساعدة تسمى wasm-pack لتسمح لنا بتحويل البرامج إلى WebAssembly بسهولة دون الحاجة إلى إضافة الأهداف والتبعيات يدويًا. ستحتاج إلى تثبيتها عبر:

```
$ cargo install wasm-pack
```

إذا كنت تستخدم Windows، قد يفشل التثبيت عند crate `openssl-sys` راجع هذا الرابط وسيتعين عليك تنزيلها يدويًا من هنا.

كما ذكرت سابقًا، سنحتاج إلى إجراء تعديل بسيط على وحدة `chip8_core` الخاصة بنا للسماح لها بالتحويل بشكل صحيح إلى هدف `wasm`. يستخدم Rust نظامًا يسمى `wasm-bindgen` لإنشاء روابط تعمل مع WebAssembly. كل الكود الذي نستخدمه من `std` جاهز بالفعل، ولكننا نستخدم أيضًا crate `rand` في الواجهة الخلفية، وهي غير مهيأة للعمل بشكل صحيح. لحسن الحظ، تدعم هذه الوظيفة، نحتاج فقط إلى تمكينها. في ملف `chip8_core/Cargo.toml` نحتاج إلى تغيير:

```
[dependencies]
rand = "^0.7.3"
```

إلى:

```
[dependencies]
rand = { version = "^0.7.3", features = ["wasm-bindgen"] }
```

كل ما يفعله هذا هو تحديد أننا سنحتاج إلى تضمين ميزة `wasm-bindgen` في crate `rand` عند التحويل، مما يسمح لها بالعمل بشكل صحيح في ملف WebAssembly الثنائي.

ملاحظة: بين وقت كتابة كود هذا الدليل وإنهاء الكتابة، تم تحديث crate `rand` إلى الإصدار 0.8. من بين التغييرات الأخرى، تمت إزالة ميزة `wasm-bindgen`. إذا كنت

ترغب في استخدام أحدث إصدار من crate `rand`، يبدو أن دعم WebAssembly تم نقله إلى crate منفصل. نظرًا لأننا نستخدم فقط الوظيفة العشوائية الأساسية، لم أشعر بالحاجة إلى الترقية إلى الإصدار 0.8، ولكن إذا كنت ترغب في ذلك، يبدو أن التكامل الإضافي مطلوب.

هذه هي المرة الأخيرة التي ستحتاج فيها إلى تعديل وحدة `chip8_core` الخاصة بك، كل شيء آخر سيتم في الواجهة الأمامية الجديدة. لنقم بإعداد ذلك الآن. أولاً، لننشئ وحدة Rust جديدة عبر:

```
$ cargo init wasm --lib
```

قد تبدو هذه الأوامر مألوفة، حيث ستقوم بإنشاء مكتبة Rust جديدة تسمى `wasm`. تمامًا مثل `desktop`، سنحتاج إلى تعديل `wasm/Cargo.toml` للإشارة إلى مكان وجود `chip8_core`.

```
[dependencies]
chip8_core = { path = "../chip8_core" }
```

الآن، الفرق الكبير بين `desktop` وواجهة `wasm` الجديدة هو أن `desktop` كان مشروعًا تنفيذيًا، حيث كان يحتوي على `main.rs` والذي كنا نحمله ونشغله. `wasm` لن تحتوي على ذلك، فهي مخصصة للتحويل إلى ملف .wasm والذي سنحمله في صفحة ويب. ستكون صفحة الويب هي الواجهة الأمامية، لذا دعنا نضيف بعض القوالب الأساسية لصفحة HTML، فقط لبدء العمل. أنشئ مجلدًا جديدًا

يسمى `web` لحفظ الكود المخصص لصفحة الويب، ثم أنشئ `web/index.html` وأضف قالب HTML الأساسي.

```
<!DOCTYPE html>
<html>
<head>
<title>Chip-8 Emulator</title>
<meta charset="utf-8">
</head>
<body>
<h1>My Chip-8 Emulator</h1>
</body>
</html>
```

سنضيف المزيد لاحقًا، ولكن هذا يكفي حاليًا. لن يعمل برنامج الويب الخاص بنا إذا قمت ببساطة بفتح الملف في متصفح الويب، ستحتاج إلى بدء خادم ويب أولاً. إذا كان لديك Python 3 مثبتًا، وهو موجود في جميع أجهزة Mac الحديثة والعديد من توزيعات Linux، يمكنك ببساطة بدء خادم ويب عبر:

```
$ python3 -m http.server
```

انتقل إلى `localhost` في متصفح الويب الخاص بك. إذا قمت بتشغيل هذا في مجلد `web`، يجب أن ترى صفحة `index.html` معروضة. لقد حاولت العثور على طريقة بسيطة ومدمجة لبدء خادم ويب محلي على Windows، ولم أجد واحدة. أنا شخصيًا أستخدم Python 3، ولكنك مرحب باستخدام أي خدمة مشابهة أخرى، مثل `npm` أو حتى بعض إضافات Visual Studio Code. لا يهم أي منها، طالما يمكنها استضافة صفحة ويب محلية.

تعريف واجهة برمجة تطبيقات WebAssembly

لدينا `chip8_core` جاهزة بالفعل، ولكننا نفتقد الآن جميع الوظائف التي أضفناها إلى `desktop`. تحميل ملف، التعامل مع ضغطات المفاتيح، إخبارها بالتنفيذ، إلخ. من ناحية أخرى، لدينا صفحة ويب (ستعمل) باستخدام JavaScript، والتي تحتاج إلى التعامل مع مدخلات المستخدم وعرض العناصر. وحدة `wasm` الخاصة بنا هي ما يقع في المنتصف. ستأخذ المدخلات من JavaScript وتحولها إلى أنواع البيانات المطلوبة من قبل `chip8_core`.

الأهم من ذلك، نحتاج أيضًا إلى إنشاء كائن `chip8_core::Emu` والحفاظ عليه في النطاق طوال فترة عمل صفحة الويب.

للبدء، دعنا نضمّن بعض الحزم الخارجية التي سنحتاجها للسماح لـ Rust بالتفاعل مع JavaScript. افتح `wasm/Cargo.toml` وأضف التبعيات التالية:

```
[dependencies]
chip8_core = { path = "../chip8_core" }
js-sys = "^0.3.46"
wasm-bindgen = "^0.2.69"

[dependencies.web-sys]
version = "^0.3.46"
features = []
```

ستلاحظ أننا نتعامل مع `web-sys` بشكل مختلف عن التبعيات الأخرى. تم تنظيم هذه الحزمة بطريقة تجعلنا بدلًا من الحصول على كل ما تحتويه ببساطة عن طريق تضمينها في `Cargo.toml`، نحتاج أيضًا إلى تحديد "ميزات" إضافية تأتي مع

الحزمة، ولكنها غير متاحة بشكل افتراضي. ابق هذا الملف مفتوحًا، حيث سنضيف إلى ميزات `web_sys` قريبًا.

نظرًا لأن هذه الحزمة ستتفاعل مع لغة أخرى، نحتاج إلى تحديد كيفية التواصل بينهما. دون الخوض في التفاصيل، يمكن لـ Rust استخدام ABI لغة C للتواصل بسهولة مع اللغات الأخرى التي تدعمه، وسيُبسّط بشكل كبير ملف wasm الثنائي الخاص بنا للقيام بذلك. لذا، سنحتاج إلى إخبار `cargo` باستخدامه. أضف هذا أيضًا في `wasm/Cargo.toml`:

```
[lib]
crate-type = ["cdylib"]
```

ممتاز. الآن إلى `wasm/src/lib.rs`. لنقم بإنشاء struct سيحتوي على كائن `Emu` الخاص بنا بالإضافة إلى جميع وظائف الواجهة الأمامية التي نحتاجها للتفاعل مع JavaScript والتشغيل. سنحتاج أيضًا إلى تضمين جميع العناصر العامة من `chip8_core`.

```
use chip8_core::*;
use wasm_bindgen::prelude::*;

#[wasm_bindgen]
pub struct EmuWasm {
    chip8: Emu,
}
```

لاحظ العلامة `#[wasm_bindgen]`، والتي تخبر المترجم أن هذا الـ struct يحتاج إلى التهيئة لـ WebAssembly. أي دالة أو struct سيتم استدعاؤها من داخل JavaScript سيحتاج إلى وجودها. دعنا نحدد المُنشئ أيضًا.

```rust
use chip8_core::*;
use wasm_bindgen::prelude::*;

#[wasm_bindgen]
pub struct EmuWasm {
    chip8: Emu,
}

#[wasm_bindgen]
impl EmuWasm {
#[wasm_bindgen(constructor)]
    pub fn new() -> EmuWasm {
        EmuWasm {
            chip8: Emu::new(),
        }
    }
}
```

بسيط جدًا. أهم شيء يجب ملاحظته هو أن الدالة `new` تتطلب تضمين `constructor` الخاص حتى يعرف المترجم ما نحاول القيام به.

الآن لدينا struct يحتوي على كائن محاكاة `chip8` الأساسي. هنا، سننفذ نفس الطرق التي احتجناها في الواجهة الأمامية لـ `desktop`، مثل تمرير ضغطات/إفلاتات المفاتيح إلى النواة، تحميل ملف، والتنفيذ. لنبدأ بتنفيذ CPU والموقتات، حيث أنها الأسهل.

```rust
#[wasm_bindgen]
impl EmuWasm {
// -- الكود غير المتغير محذوف --

#[wasm_bindgen]
pub fn tick(&mut self) {
    self.chip8.tick();
}

#[wasm_bindgen]
```

```
    pub fn tick_timers(&mut self) {
        self.chip8.tick_timers();
    }
}
```

هذا كل شيء، هذه مجرد أغلفة رقيقة لاستدعاء الدوال المقابلة في `chip8_core`. هذه الدوال لا تأخذ أي مدخلات، لذا لا يوجد شيء معقد عليها سوى التنفيذ.

تذكر دالة `reset` التي أنشأناها في `chip8_core`، ولكننا لم نستخدمها أبدًا؟ حسنًا، سنستخدمها الآن. ستكون هذه مجرد غلاف مثل الدوال السابقة.

```
    #[wasm_bindgen]
    pub fn reset(&mut self) {
        self.chip8.reset();
    }
```

ضغط المفاتيح هو أول هذه الدوال التي ستنحرف عما تم في `desktop`. يعمل هذا بطريقة مشابهة لما فعلناه في `desktop`، ولكن بدلًا من أخذ ضغطة مفتاح SDL، سنحتاج إلى قبول واحدة من JavaScript. لقد وعدت بإضافة بعض ميزات `web-sys`، لذا لنفعل ذلك الآن. عد إلى `wasm/Cargo.toml` وأضف ميزة `KeyboardEvent`.

```
[dependencies.web-sys]
    version = "^0.3.46"
        features = [
    "KeyboardEvent"
        ]
```

```
use web_sys::KeyboardEvent;

// -- الكود غير المتغير محذوف --

impl EmuWasm {
// -- الكود غير المتغير محذوف --
```

```
                                        #[wasm_bindgen]
pub fn keypress(&mut self, evt: KeyboardEvent, pressed: bool
                                    let key = evt.key();
                                if let Some(k) = key2btn(&key) {
                                self.chip8.keypress(k, pressed);
                                                        }
                                                        }
                                                        }

                                fn key2btn(key: &str) -> Option<usize> {
                                                    match key {
                                                "1" => Some(0x1),
                                                "2" => Some(0x2),
                                                "3" => Some(0x3),
                                                "4" => Some(0xC),
                                                "q" => Some(0x4),
                                                "w" => Some(0x5),
                                                "e" => Some(0x6),
                                                "r" => Some(0xD),
                                                "a" => Some(0x7),
                                                "s" => Some(0x8),
                                                "d" => Some(0x9),
                                                "f" => Some(0xE),
                                                "z" => Some(0xA),
                                                "x" => Some(0x0),
                                                "c" => Some(0xB),
                                                "v" => Some(0xF),
                                                     _ =>   None,
                                                        }
                                                        }
```

هذا مشابه جدًا لتنفيذنا في `desktop`، إلا أننا سنأخذ حدث `KeyboardEvent` من JavaScript، والذي سيؤدي إلى سلسلة نصية لتحليلها. لاحظ أن سلاسل المفاتيح حساسة لحالة الأحرف، لذا حافظ على كل شيء بأحرف صغيرة إلا إذا كنت تريد من اللاعبين الضغط على Shift كثيرًا.

قصة مشابهة تنتظرنا عند تحميل لعبة، ستتبع نمطًا مشابهًا، إلا أننا سنحتاج إلى استقبال ومعالجة كائن JavaScript.

```
use js_sys::Uint8Array;
// -- الكود غير المتغير محذوف --

impl EmuWasm {
// -- الكود غير المتغير محذوف --

#[wasm_bindgen]
pub fn load_game(&mut self, data: Uint8Array) {
self.chip8.load(&data.to_vec());
}
}
```

الشيء الوحيد المتبقي هو دالة الرسم الفعلية على الشاشة. سأقوم بإنشاء دالة فارغة هنا، ولكننا سنؤجل تنفيذها حاليًا، بدلًا من ذلك سنوجه انتباهنا مرة أخرى إلى صفحة الويب، ونبدأ العمل من الاتجاه الآخر.

```
impl EmuWasm {
// -- الكود غير المتغير محذوف --

#[wasm_bindgen]
pub fn draw_screen(&mut self, scale: usize) {
// TODO
}
}
```

سنعود إلى هنا بمجرد إعداد JavaScript الخاص بنا ومعرفة كيفية الرسم بالضبط.

إنشاء وظائف الواجهة الأمامية

حان الوقت للتعمق في JavaScript. أولاً، دعنا نضيف بعض العناصر الإضافية إلى صفحة الويب البسيطة جدًا. عندما أنشأنا المحاكي للتشغيل على جهاز كمبيوتر، استخدمنا SDL لإنشاء نافذة للرسم عليها. بالنسبة لصفحة الويب، سنستخدم عنصرًا يوفره لنا HTML5 يسمى *canvas*. سنقوم أيضًا بالإشارة إلى صفحة الويب الخاصة بنا إلى النص البرمجي JS (غير الموجود حاليًا).

```
<!DOCTYPE html>
<html>
<head>
<title>Chip-8 Emulator</title>
<meta charset="utf-8">
</head>
<body>
<h1>My Chip-8 Emulator</h1>
<label for="fileinput">Upload a Chip-8 game: </label>
<input type="file" id="fileinput" autocomplete="off"/>
<br/>
<canvas id="canvas">If you see this message, then your b
</body>
<script type="module" src="index.js"></script>
</html>
```

أضفنا هنا ثلاثة أشياء، أولاً زرًا يسمح للمستخدمين باختيار لعبة Chip-8 لتشغيلها عند النقر عليه. ثانيًا، عنصر `canvas`، والذي يتضمن رسالة قصيرة لأي مستخدمين غير محظوظين لديهم متصفح قديم. أخيرًا أخبرنا صفحة الويب بتحميل النص البرمجي `index.js` الذي سنقوم بإنشائه. لاحظ أنه في وقت كتابة هذا الدليل، من أجل تحميل ملف .wasm عبر JavaScript، تحتاج إلى تحديد أنه من نوع `module`.

الآن، لنقم بإنشاء `index.js` ونحدد بعض العناصر التي سنحتاجها. أولاً، نحتاج إلى إخبار JavaScript بتحميل وظائف WebAssembly. الآن، لن نقوم بتحميلها مباشرة هنا. عندما نستخدم `wasm-pack` للتحويل، سيتم إنشاء ليس فقط ملف wasm. الخاص بنا، ولكن أيضًا "صمغ" JavaScript الذي سيلف كل دالة قمنا بتعريفها حول دالة JavaScript يمكننا استخدامها هنا.

```
import init, * as wasm from "./wasm.js"
```

هذا يستورد جميع وظائفنا، بالإضافة إلى دالة خاصة `init` سيحتاج إلى استدعائها أولاً قبل أن نتمكن من استخدام أي شيء من `wasm`.

لنقم بتعريف بعض الثوابت وإجراء بعض الإعدادات الأساسية الآن.

```
import init, * as wasm from "./wasm.js"

const WIDTH = 64
const HEIGHT = 32
const SCALE = 15
const TICKS_PER_FRAME = 10
let anim_frame = 0

const canvas = document.getElementById("canvas")
canvas.width = WIDTH * SCALE
canvas.height = HEIGHT * SCALE

const ctx = canvas.getContext("2d")
ctx.fillStyle = "black"
ctx.fillRect(0, 0, WIDTH * SCALE, HEIGHT * SCALE)

const input = document.getElementById("fileinput")
```

كل هذا سيبدو مألوفًا من بناء `desktop` الخاص بنا. نحن نحصل على لوحة HTML ونضبط حجمها إلى أبعاد شاشة Chip-8 الخاصة بنا، بالإضافة إلى تكبيرها قليلاً (لا تتردد في تعديل هذا وفقًا لتفضيلاتك).

لنقم بإنشاء وظيفة رئيسية `run` والتي ستحمّل كائن `EmuWasm` الخاص بنا وتتعامل مع المحاكاة الرئيسية.

```
async function run() {
    await init()
let chip8 = new wasm.EmuWasm()

document.addEventListener("keydown", function(evt) {
    chip8.keypress(evt, true)
})

document.addEventListener("keyup", function(evt) {
    chip8.keypress(evt, false)
})

input.addEventListener("change", function(evt) {
    // التعامل مع تحميل الملف
}, false)
}

run().catch(console.error)
```

هنا، قمنا باستدعاء الوظيفة الإلزامية `init` التي تخبر متصفحنا بتهيئة ثنائي WebAssembly قبل أن نستخدمه. ثم نقوم بإنشاء محاكي الخلفية الخاص بنا عن طريق إنشاء كائن جديد `EmuWasm`.

سنقوم الآن بالتعامل مع تحميل ملف عند الضغط على الزر.

```
input.addEventListener("change", function(evt) {
// إيقاف عرض اللعبة السابقة، إذا كانت موجودة
if (anim_frame != 0) {
window.cancelAnimationFrame(anim_frame)
}

let file = evt.target.files[0]
if (!file) {
alert("فشل في قراءة الملف")
return
}

// تحميل اللعبة كـ Uint8Array، إرسالها إلى wasm. بدء الحلقة الرئيسية
let fr = new FileReader()
fr.onload = function(e) {
let buffer = fr.result
const rom = new Uint8Array(buffer)
chip8.reset()
chip8.load_game(rom)
mainloop(chip8)
}
fr.readAsArrayBuffer(file)
}, false)

function mainloop(chip8) {
}
```

تضيف هذه الوظيفة مستمع حدث إلى زر `input` الخاص بنا والذي يتم تشغيله عند النقر عليه. استخدم واجهة `desktop` الأمامية SDL لإدارة الرسم في النافذة، وكذلك للتأكد من أننا نعمل بسرعة 60 إطارًا في الثانية. الميزة المماثلة للوحات هي "إطارات الرسوم المتحركة". في أي وقت نريد عرض شيء ما على اللوحة، نطلب من النافذة تحريك إطار، وسوف تنتظر حتى ينقضي الوقت الصحيح لضمان أداء 60 إطارًا في الثانية. سنرى كيف يعمل هذا في لحظة، ولكن الآن، نحتاج إلى إخبار برنامجنا أنه إذا كنا نحمّل لعبة جديدة، نحتاج إلى إيقاف الرسوم

المتحركة السابقة. سنقوم أيضًا بإعادة تعيين المحاكي قبل تحميل ROM، للتأكد من أن كل شيء كما بدأ، دون الحاجة إلى إعادة تحميل صفحة الويب.

بعد ذلك، ننظر إلى الملف الذي أشار إليه المستخدم. لا نحتاج إلى التحقق مما إذا كان برنامج Chip-8 فعليًا، ولكننا نحتاج إلى التأكد من أنه ملف من نوع ما. ثم نقوم بقراءته وتمريره إلى الخلفية عبر كائن `EmuWasm` الخاص بنا. بمجرد تحميل اللعبة، يمكننا القفز إلى حلقة المحاكاة الرئيسية!

```
function mainloop(chip8) {
    // الرسم فقط كل بضع دورات
    for (let i = 0; i < TICKS_PER_FRAME; i++) {
        chip8.tick()
    }
    chip8.tick_timers()

    // مسح اللوحة قبل الرسم
    ctx.fillStyle = "black"
    ctx.fillRect(0, 0, WIDTH * SCALE, HEIGHT * SCALE)
    // إعادة تعيين لون الرسم إلى الأبيض قبل عرض الإطار
    ctx.fillStyle = "white"
    chip8.draw_screen(SCALE)

    anim_frame = window.requestAnimationFrame(() => {
        mainloop(chip8)
    })
}
```

يجب أن يبدو هذا مشابهًا جدًا لما فعلناه لواجهة `desktop` الأمامية. نقوم بالتنفيذ عدة مرات قبل مسح اللوحة وإخبار كائن `EmuWasm` الخاص بنا برسم الإطار الحالي على اللوحة. هنا نخبر النافذة أننا نرغب في عرض إطار، ونحتفظ بمعرفها إذا احتجنا إلى إلغائه أعلاه. سوف ينتظر `requestAnimationFrame` لضمان أداء 60

إطارًا في الثانية، ثم يعيد تشغيل `mainloop` عندما يحين الوقت، ويبدأ العملية من جديد.

تجميع ثنائي WebAssembly الخاص بنا

قبل أن نذهب أبعد من ذلك، دعنا نحاول بناء كود Rust الخاص بنا ونتأكد من أنه يمكن تحميله بواسطة صفحة الويب دون مشاكل. سيتعامل `wasm-pack` مع تجميع ثنائي .wasm، ولكننا نحتاج أيضًا إلى تحديد أننا لا نرغب في استخدام أي أنظمة حزم ويب مثل `npm`. للبناء، قم بتغيير الدلائل إلى مجلد `wasm` وقم بتشغيل:

```
$ wasm-pack build --target web
```

بمجرد اكتماله، سيتم بناء الأهداف في دليل جديد `pkg`. هناك عدة عناصر هنا، ولكن الوحيدة التي نحتاجها هي `wasm_bg.wasm` و `wasm_bg.wasm`. `wasm.js` هو مزيج من حزمتي `wasm` و Rust `chip8_core` المترجمة في واحدة، و `wasm.js` هو "الغراء" JavaScript الذي أدرجناه سابقًا. إنه في الغالب أغلفة حول API الذي حددناه في `wasm` بالإضافة إلى بعض كود التهيئة. إنه في الواقع قابل للقراءة إلى حد ما، لذا فإن الأمر يستحق النظر إلى ما يفعله.

تشغيل الصفحة في خادم ويب محلي يجب أن يسمح لك باختيار وتحميل لعبة دون ظهور أي تحذيرات في وحدة تحكم المتصفح. ومع ذلك، لم نكتب وظيفة عرض الشاشة بعد، لذا دعنا ننهي ذلك حتى نتمكن من رؤية لعبتنا تعمل بالفعل.

الرسم على اللوحة

هذه هي الخطوة الأخيرة، العرض على الشاشة. لقد أنشأنا وظيفة `draw_screen` فارغة في كائن `EmuWasm` الخاص بنا، ونستدعيها في الوقت المناسب، ولكنها حاليًا لا تفعل أي شيء. الآن، هناك طريقتان يمكننا التعامل مع هذا. يمكننا إما تمرير إطار العرض إلى JavaScript وعرضه، أو يمكننا الحصول على لوحتنا في ثنائي `EmuWasm` الخاص بنا وعرضها في Rust. أي من الطريقتين ستكون جيدة، ولكن شخصيًا وجدت أن التعامل مع العرض في Rust أسهل.

لقد استخدمنا الحزمة `web_sys` للتعامل مع أحداث `KeyboardEvent` في JavaScript في Rust، ولكن لديها وظائف لإدارة العديد من عناصر JavaScript الأخرى. مرة أخرى، تلك التي نرغب في استخدامها تحتاج إلى تعريفها كسمات في `wasm/Cargo.toml`.

```
[dependencies.web-sys]
   version = "^0.3.46"
          features = [
"CanvasRenderingContext2d",
           "Document",
            "Element",
       "HtmlCanvasElement",
           "ImageData",
        "KeyboardEvent",
            "Window"
                ]
```

هذه نظرة عامة على خطواتنا التالية. من أجل العرض على لوحة HTML5، تحتاج إلى الحصول على كائن اللوحة *وسياقها* وهو الكائن الذي يتم استدعاء وظائف الرسم عليه. نظرًا لأن ثنائي WebAssembly الخاص بنا تم تحميله بواسطة صفحة

الويب الخاصة بنا، فإنه يمكنه الوصول إلى جميع عناصرها تمامًا كما يفعل نص JS. سنقوم بتغيير المُنشئ new للحصول على النافذة الحالية، اللوحة، والسياق كما تفعل في JavaScript.

```
                                                use wasm_bindgen::JsCast;
use web_sys::{CanvasRenderingContext2d, HtmlCanvasElement, Keybo

                                                    #[wasm_bindgen]
                                                pub struct EmuWasm {
                                                        chip8: Emu,
                                                ctx: CanvasRenderingContext2d,
                                                                }

                                                    #[wasm_bindgen]
                                                     impl EmuWasm {
                                                #[wasm_bindgen(constructor)]
                                        pub fn new() -> Result<EmuWasm, JsValue> {
                                                        let chip8 = Emu::new();

            let document = web_sys::window().unwrap().document().unw
            let canvas = document.get_element_by_id("canvas").unwrap
                                    let canvas: HtmlCanvasElement = canvas
                                          .dyn_into::<HtmlCanvasElement>()
                                                       .map_err(|_| ())
                                                             .unwrap();

                                                let ctx = canvas.get_context("2d")
                                                             .unwrap().unwrap()
                                                .dyn_into::<CanvasRenderingContext2d>()
                                                             .unwrap();

                                                    Ok(EmuWasm{chip8, ctx})
                                                                }

                                                    // -- كود غير متغير محذوف --
                                                                }
```

يجب أن يبدو هذا مألوفًا لأولئك الذين قاموا ببرمجة JavaScript من قبل. نحن نحصل على لوحة النافذة الحالية ونحصل على سياقها ثنائي الأبعاد، والذي يتم حفظه كمتغير عضو في بنية `EmuWasm` الخاصة بنا. الآن بعد أن أصبح لدينا سياق فعلي للرسم عليه، يمكننا تحديث وظيفة `draw_screen` للرسم عليه.

```
                                    #[wasm_bindgen]
            pub fn draw_screen(&mut self, scale: usize) {
                    let disp = self.chip8.get_display();
             for i in 0..(SCREEN_WIDTH * SCREEN_HEIGHT) {
                                        if disp[i] {
                                let x = i % SCREEN_WIDTH;
                                let y = i / SCREEN_WIDTH;
                                    self.ctx.fill_rect(
                                    (x * scale) as f64,
                                    (y * scale) as f64,
                                          scale as f64,
                                           scale as f64
                                                );
                                                 }
                                                 }
                                                 }
```

نحصل على مخزن عرض من `chip8_core` الخاص بنا ونكرر عبر كل بكسل. إذا تم تعيينه، نرسمه مكبرًا إلى القيمة التي تم تمريرها من واجهتنا الأمامية. لا تنس أننا قمنا بالفعل بمسح اللوحة إلى الأسود وتعيين لون الرسم إلى الأبيض قبل استدعاء `draw_screen`، لذا لا يحتاج إلى القيام بذلك هنا.

هذا كل شيء! التنفيذ انتهى. كل ما تبقى هو بناؤه وتجربته بأنفسنا.

أعد البناء عن طريق الانتقال إلى دليل `wasm` وتشغيل:

```
$ wasm-pack build --target web
  $ mv pkg/wasm_bg.wasm ../web
       $ mv pkg/wasm.js ../web
```

الآن ابدأ خادم الويب الخاص بك واختر لعبة. إذا سار كل شيء على ما يرام، يجب أن تكون قادرًا على لعب ألعاب Chip-8 في المتصفح بنفس جودة سطح المكتب!

جدول Opcodes

فهم عمود Opcode:

- أي أرقام ست عشرية (0-9، A-F) تظهر في opcode يتم تفسيرها حرفيًا، وتستخدم لتحديد العملية المطلوبة.
- الرمز البريدي X أو Y يستخدم القيمة المخزنة في VX/VY.
- N يشير إلى قيمة ست عشرية حرفية. NN أو NNN تشير إلى رقمين أو ثلاثة أرقام ست عشرية على التوالي.

مثال: التعليمات 0xD123 ستطابق مع opcode `DXYN`، حيث VX هو V1، VY هو V2، و N هو 3 (ارسم شريطًا بحجم 8x3 عند (V1, V2)).

Opcode	الوصف	الملاحظات
0000	Nop	لا تفعل شيئًا، انتقل إلى opcode التالي
00E0	مسح الشاشة	
00EE	العودة من subroutine	
1NNN	القفز إلى العنوان 0xNNN	
2NNN	استدعاء 0xNNN	ادخل subroutine عند 0xNNN، أضف PC الحالي إلى المكدس حتى نتمكن من العودة هنا

Opcode	الوصف	الملاحظات
3XNN	تخطي إذا كان == VX 0xNN	
4XNN	تخطي إذا كان 0xNN =! VX	
5XY0	تخطي إذا كان VX == VY	
6XNN	VX = 0xNN	
7XNN	VX += 0xNN	لا يؤثر على علم الحمل
8XY0	VX = VY	
8XY1	VX |= VY	
8XY2	VX &= VY	
8XY3	VX ^= VY	
8XY4	VX += VY	يضع VF إذا كان هناك حمل
8XY5	VX -= VY	يمسح VF إذا كان هناك استلاف
8XY6	VX >>= 1	تخزين البت المسقط في VF
8XY7	VX = VY - VX	يمسح VF إذا كان هناك استلاف
8XYE	VX <<= 1	تخزين البت المسقط في VF
9XY0	تخطي إذا كان VY =! VX	
ANNN	I = 0xNNN	
BNNN	القفز إلى 0xNNN + V0	
CXNN	VX = rand() & 0xNN	
DXYN	رسم شريط عند (VX, VY)	الشريط بطول 0xN بكسل، تشغيل/إطفاء بناءً على القيمة في I، يتم تعيين VF إذا

Opcode	الوصف	الملاحظات
		تم قلب أي بكسل
EX9E	تخطي إذا كان مفتاح الفهرس في VX مضغوط	
EXA1	تخطي إذا كان مفتاح الفهرس في VX غير مضغوط	
FX07	VX = مؤخر الوقت	
FX0A	ينتظر ضغط مفتاح، يخزن الفهرس في VX	عملية حظر
FX15	مؤخر الوقت = VX	
FX18	مؤخر الصوت = VX	
FX1E	I += VX	
FX29	تعيين I إلى عنوان حرف الخط في VX	
FX33	تخزين ترميز BCD لـ VX في I	
FX55	تخزين V0 حتى VX في عنوان RAM بدءًا من I	نطاق شامل

Opcode	الوصف	الملاحظات
FX65	ملء V0 حتى VX بقيم RAM بدءًا من العنوان في I	شامل

سجل التغييرات

الإصدار 1.0

- الإصدار الأولي
- يتضمن الكود المصدري لمحاكي Chip-8 بالإضافة إلى مستند PDF يناقش تطويره

الإصدار 1.01

- إصلاح خطأ مطبعي: المكدس هو نظام LIFO، وليس FIFO
- إضافة إصلاحات أخطاء مفقودة من الكود المصدري إلى الكتاب
- إضافة توليد ePub

الإصدار 1.1

- تم تحديث الكثير من النص لتحسين التدفق (شكر خاص لـ KDR للتحرير).